UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Departamento de Automática

Nuevas técnicas de inyección de fallos en sistemas embebidos mediante el uso de modelos virtuales descritos en el nivel de transacción

Tesis Doctoral

Antonio da Silva Fariña

Alcalá de Henares, Enero 2015

Universidad de Alcalá

DEPARTAMENTO DE AUTOMÁTICA
Edificio Politécnico
28871 Alcalá de Henares (Madrid)
Tel: 91 885 65 94. Fax: 91 885 69 23
secre@aut.uah.es

Dr. D. Agustín Martínez Hellín, Profesor Titular de la Universidad de Alcalá y Director del Departamento de Automática

INFORMA: Que la Tesis Doctoral titulada "**Nuevas técnicas de inyección de fallos en sistemas embebidos mediante el uso de modelos virtuales descritos en el nivel de transacción**", presentado por D. Antonio da Silva Fariña, y dirigida por el Dr. D. Sebastián Sánchez Prieto, cumple con todos los requisitos científicos y metodológicos para ser defendida ante un tribunal.

Alcalá de Henares, 19 de diciembre de 2014

Fdo.: Agustín Martínez Hellín
Director del Departamento de Automática

SPACE RESEARCH GROUP (SRG)
Edificio Politécnico
28805 Alcalá de Henares (Madrid)
Teléfonos: 91 885 65 94
Fax: 91 885 69 23
secre@aut.uah.es

SEBASTIÁN SÁNCHEZ PRIETO, Titular de Universidad del área de Arquitectura y Tecnología de Computadores de la Universidad de Alcalá,

HACE CONSTAR

Que el trabajo "**Nuevas técnicas de inyección de fallos en sistemas embebidos mediante el uso de modelos virtuales descritos en el nivel de transacción**", presentado por D. Antonio da Silva Fariña, ha sido realizado en el Departamento de Automática bajo mi dirección, reuniendo los méritos suficientes para optar al grado de doctor, por lo que puede proceder a su depósito y lectura.

Alcalá de Henares, 19 de diciembre de 2014

Fdo.: Dr. Sebastián Sánchez Prieto

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Departamento de Automática

Nuevas técnicas de inyección de fallos en sistemas embebidos mediante el uso de modelos virtuales descritos en el nivel de transacción

Tesis Doctoral

Antonio da Silva Fariña

Director

Dr. Sebastián Sánchez Prieto

Alcalá de Henares, Enero 2015

Estoy contento de haberlo hecho, en parte porque valía la pena,
pero sobretodo, porque no tendré que hacerlo otra vez
Mark Twain

Tal vez no haya llegado donde pretendía ir, pero creo
que he acabado donde necesitaba estar
Douglas Adams

Dedicatoria

En memoria de Julia Rosa da Silva.

Que no aprendió a leer ni escribir, pero que tenía en sus dedos una capacidad de cálculo equivalente a un coprocesador matemático.

A Laurindinha

'O laurindinha
Vem 'a janela
Ver o teu amor
Ai ai ai que ele vai para a guerra

Se ele vai para a guerra
Deixai-o ir
Ele 'e rapaz novo
Ai ai ai ele torna a vir

Agradecimientos

Sin duda, la cantidad de servidumbres que uno va acumulando a lo largo de su vida es bastante más grande que la de privilegios. Ello implica que es imposible reconocer en unas pocas líneas a todas las personas e instituciones con las que se está en deuda y que merecerían ser mencionadas.

No quisiera dejar de mencionar la institución educativa gracias a la cual pude acceder a la educación media y superior. Durante muchos años, las Universidades Laborales ofrecieron un sistema de cobertura educativa que hizo real el derecho a la educación de la población con menos recursos. He tenido el placer de estudiar en dos centros, Ourense y Alcalá de Henares, y solo tengo palabras de agradecimiento para el conjunto de la institución.

La referencia a mis padres es obligada. Sin su esfuerzo y dedicación desinteresada no estaría hoy escribiendo la memoria de una tesis doctoral y tal vez ahora pueda poner en práctica el último consejo de mi madre: «Hijo, no estudies más que te vas a volver loco». Gracias también a Blanca por su paciencia y por preguntar N veces al día, con $N \to \infty$, «¿Cuando acabas la tesis?»

Quiero agradecer al Grupo de Investigación del Espacio (SRG) en su conjunto y a Sebastián Sánchez, mi tutor y amigo, en particular, la oportunidad de realizar este trabajo de investigación en el marco de un proyecto de vanguardia como Solar Orbiter.

Por último, agradecerte a ti, que estás leyendo estas líneas, el interés en este trabajo. Da un poco igual la razón por la que la te hayas aproximado a él. Si al leerlo, has encontrado algo que te haya sido de utilidad, me alegro. En caso contrario, no desesperes y sigue buscando. Yo, también lo haré...

Este trabajo ha sido financiado por el MICINN, ahora Ministerio de Economía y Competitividad, en el marco de los proyectos AYA2011-29727-C02-02, AYA2012-39810-C02-02 y ESP2013-48346-C2-2-R.

Resumen

Mejor software y más rápido. Este es el desafío que se deriva de la necesidad de construir sistemas cada vez más *inteligentes*. En cualquier diseño embebido actual, el software es un componente fundamental que dota al sistema de una alta capacidad de configuración, gran número de funcionalidades y elasticidad en el comportamiento del sistema en situaciones excepcionales. Si además el desarrollo del conjunto hardware/software integrado en un *System on Chip* (SoC), forma parte de un sistema de control crítico donde se deben tener en cuenta requisitos de tolerancia a fallos, la verificación exhaustiva de los mismos consume un porcentaje cada vez más importante de los recursos totales dedicados al desarrollo y puesta en funcionamiento del sistema. En este contexto, el uso de metodologías clásicas de codiseño y coverificación es completamente ineficiente, siendo necesario el uso de nuevas tecnologías y herramientas para el desarrollo y verificación tempranos del software embebido. Entre ellas se puede incluir la propuesta en este trabajo de tesis, la cual aborda el problema mediante el uso de modelos ejecutables del hardware definidos en el nivel de transacción.

Debido a los estrictos requisitos de robustez que imperan en el desarrollo de software espacial, es necesario llevar a cabo tareas de verificación en etapas muy tempranas del desarrollo para asegurar que los mecanismos de tolerancia a fallos, avanzados en la especificación del sistema, funcionan adecuadamente. De forma general, es deseable que estas tareas se realicen en paralelo con el desarrollo hardware, anticipando problemas o errores existentes en la especificación del sistema. Además, la verificación completa de los mecanismos de excepción implementados en el software, puede ser imposible de realizar en hardware real ya que los escenarios de fallo deben ser artificial y sistemáticamente generados mediante técnicas de inyección de fallos que permitan realizar campañas de inyección controlables, observables y reproducibles.

En esta tesis se describe la investigación, desarrollo y uso de una plataforma virtual denominada "Leon2ViP", con capacidad de inyección de fallos y basada en interfaces SystemC/TLM2 para el desarrollo temprano y verificación de software embebido en el marco del proyecto Solar Orbiter. De esta forma ha sido posible ejecutar y probar exactamente el mismo código binario a ejecutar en el hardware real, pero en un entorno más controlable y determinista. Ello permite la realización de campañas de inyección de fallos muy focalizadas que no serían posible de otra manera. El uso de "Leon2ViP" ha significado una mejora significante, en términos de coste y tiempo, en el desarrollo y verificación del software de arranque de la unidad de control del intrumento (ICU) del detector de partículas energéticas (EPD) embarcado en Solar Orbiter.

Abstract

Better software, faster. This is the challenge that stems from the need to build increasingly smarter systems. In any current embedded design, the software is a key component that provides the system with a high configuration capacity, large number of features and resilience of the system behavior in exceptional situations. If addition, if the hardware/software suite under development is part of a critical system integrated in a *System on Chip* (SoC), where fault tolerant requirements must be taken into account, a comprehensive verification of those requirements consumes an increasingly large percentage of resources. In this context, the use of traditional codesign and coverification methods are completely inefficient, requiring the use of new technologies and tools for the early development and verification of embedded software. These include the proposal made in this thesis that proposes the use of executable models of the hardware defined at transaction level.

Because of the tough robustness requirements that prevail in space software development, it is imperative to carry out verification tasks in very early development stages to ensure that the implemented exception mechanisms, identified in the specification of the system, work properly. In general, these tasks should be are carried out in parallel with the hardware development, anticipating problems or mistakes in the existing system specification. In addition, complete verification mechanisms implemented in the software exception may not be possible in real hardware real since the faulty scenarios must be artificial and systematically generated using fault injection techniques that allow controllable, observable and reproducible fault injection campaigns.

This thesis describes the research, development and use of a virtual platform called "Leon2ViP", with fault injection capabilities and based on SystemC/TLM2 interfaces for the early development and testing of embedded software within the Solar Orbiter project. This way it is posible to run the exact same target binary software as runs on the physical system in a more controlled and deterministic environment. This enables unmanned and tightly focused fault injection campaigns, not possible otherwise. The use of "Leon2ViP" has meant a signicant improvement, in both time and cost, in the development and verification processes of the Instrument Control Unit boot software on board Solar Orbiter's Energetic Particle Detector.

Índice general

Índice de figuras

Lista de acrónimos

BDM *Background Debug Mode.*

CAN *Controller Area Network.*

DBI *Dynamic Binary Instrumentation.*

DMI *Direct Memory Interface.*

ECU *Engine Control Units.*

EDAC *Error Detection and Correction.*

EGSE *Electrical Ground Support Equipment.*

FDIR *Fault Detection Isolation and Recovery.*

FMEA *Failure Mode and Effects Analysis.*

IRL *Interrupt Request Level.*

ISS *Instruction Set Simulators.*

ITRS *International Technology Roadmap for Semiconductors.*

JTAG *Joint Test Action Group.*

MEU *Multiple Event Upset.*

NIST *National Institute of Standards and Technology.*

NRE *Non-recurring Engineering.*

NSA *National Security Agency.*

OCD *On-Chip Debugger.*

OSCI *Open SystemC Initiative.*

PIM *Platform Independent Model.*

POSIX *Portable Operating System Interface.*

PSM *Platform Specific Model.*

PUS *Packet Utilisation Standard.*

RTL *Register Transfer Level.*

SDP *Software Data Protection.*

SEE *Single Event Effects.*

SEL *Single Event Latchup.*

SEU *Single Event Upset.*

SHA *Secure Hash Algorithm.*

SWIFI *Software Implemented Fault Injection.*

TID *Total Ionizing Dose.*

TLM *Transaction Level Modeling.*

VHIL *Virtual Hardware in the Loop.*

WCET *Worst Case Execution Time.*

Capítulo 1

Introducción

El software es como la entropía: difícil de atrapar, no pesa, y cumple la Segunda Ley de la Termodinámica, es decir, siempre tiende a incrementarse.
Norman Ralph Augustine, Ingeniero aeronáutico

El Software y las catedrales son prácticamente lo mismo. Primero las construimos, luego rezamos.
Sam Redwine, 4th International Software Process Workshop

1.1. Motivación de la tesis

Hoy en día cualquier persona interacciona al cabo del día con una gran cantidad de sistemas embebidos en teléfonos, coches, cajeros automáticos y en electrónica de consumo de todo tipo. El elemento común a todos ellos es la presencia de algún elemento procesador con su software asociado. Esta circunstancia ha proporcionado al diseñador de sistemas la capacidad de integrar funciones cada vez más complejas con el incremento espectacular del software embebido incluido. Los efectos laterales no deseados son la gran complejidad del desarollo y verificación de los nuevos diseños, la necesidad de integrar equipos de trabajo hardware/software, que tradicionalmente habían trabajado por separado, y el consiguiente incremento de los costes no recurrentes, *Non-recurring Engineering* (NRE) [ITR11]. Este fenomeno es conocido como la crisis de la complejidad y ha potenciado la investigación en el terreno del diseño y verificación de sistemas en chip.

El 19 de abril de 1965, la revista *Electronics* publicó un artículo escrito por Gordon Moore donde se enunció una de las leyes más conocidas de la industria de fabricación de componentes electrónicos [Moo65]. Moore, cofundador de Intel 10 años más tarde, observó una tendencia en los primeros días de la microelectrónica que básicamente anticipaba que la complejidad de los circuitos integrados se duplicaría cada año y medio. Aunque esta afirmación fue matizada por el propio Moore años más tarde, lo cierto es que desde entonces la habilidad de la industria para integrar mayor cantidad de transistores en un componente no ha dejado de crecer, aumentando en consecuencia la complejidad de los sistemas implementados con dicha lógica. Sin embargo con la llegada del nuevo siglo los dispositivos han incorporado de forma genérica un nuevo componente que ha aumentado

de forma muy significativa la complejidad de los diseños. En los últimos años, además del incremento exponencial de la capacidad lógica de los dispositivos integrados, se ha observado un incremento similar en el número de líneas de código del software embebido. Este crecimiento combinado tiene consecuencias inmediatas sobre la capacidad de diseño y verificación de los sistema en un tiempo razonable, tiempo cada vez más constreñido por razones de índole económica como el *tiempo de mercado*[1]. A día de hoy el software embebido marca la diferencia en las funcionalidades prestadas por un sistema. Ello implica que su desarrollo y verificación puede llegar a suponer un porcentaje cada vez más importante del tiempo de desarrollo del sistema completo. Si a ello se añaden requisitos de tolerancia a fallos, el proceso de verificación puede tornarse muy complejo e incluso inabordable en los plazos previstos.

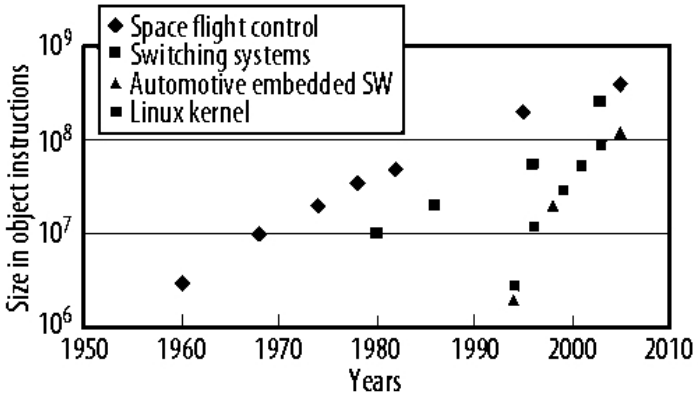


Figura 1.1: Complejidad del software embebido [EJ09]

En la figura 1.1 se puede observar que, tomando como referencia el número de líneas de código escritas en C y en lenguajes de descripción hardware como VHDL, la complejidad del software embebido ha sobrepasado la complejidad hardware. Además, la productividad software está sufriendo un incremento menor que la productividad hardware. Esta diferencia se demuestra con el hecho de que el número de personas dedicadas al desarrollo de software embebido es mayor que el dedicado a desarrollo hardware [Gro13]. Como se establece en *The Mythical Man-Month* [Bro95], no es suficiente con incrementar el número de desarroladores, sino que la reducción de la brecha vendrá de la mejora de las metodologías y las herramientas de diseño y prueba.

La brecha de productividad en diseño, es un término ampliamente conocido que define la incapacidad de las actuales metodologías/herramientas de diseño para seguir el ritmo de la industria electrónica en cuanto a capacidad de integración y aprovechamiento de todas las capacidades de los chips actuales. Siendo éste un problema importante, hay que añadir un aspecto adicional que lo complica aún más. Si la capacidad de organizar de forma coherente la lógica disponible y el software embebido en un chip crece más lentamente que la capacidad física de los componentes, la capacidad de verificar que el

[1] Tiempo transcurrido desde la definición del sistema hasta su comercialización

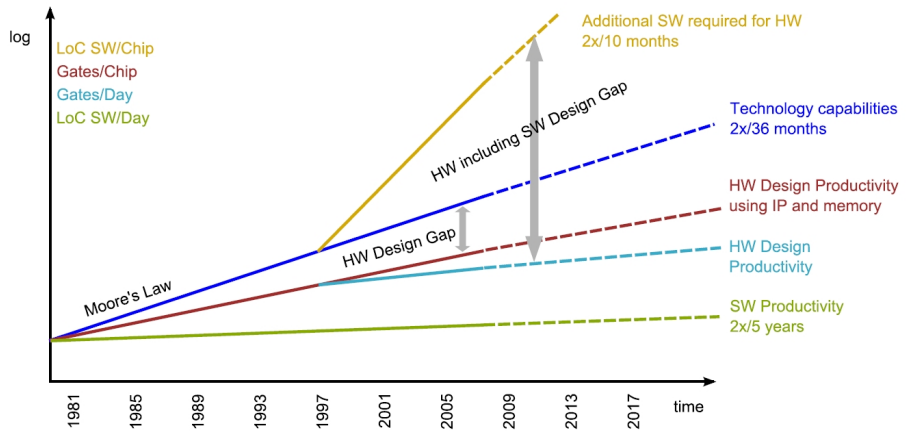


Figura 1.2: Brecha de productividad en diseño y verificación [ITR11]

diseño satisface adecuadamente los requisitos que se le han impuesto, crece a un ritmo todavía menor, véase la figura 1.2. Si además se introducen requisitos relacionados con la tolerancia a fallos, robustez y/o elasticidad (resilencia) del sistema, la brecha de productividad en diseño y verificación añade nuevos desafíos al desarrollo de sistemas críticos. Ello desemboca en la necesidad de nuevas metodologías/herramientas de diseño, desarrollo y verificación completa de los sistemas. De acuerdo con *International Technology Roadmap for Semiconductors* (ITRS), la tendencia revela un crecimiento constante en la cantidad de esfuerzo empleado en verificación, véase la figura 1.3.

1.2. Planteamiento del problema

Si el problema relacionado con la verificación de sistemas complejos es importante en circustancias normales, se torna definitivo cuando el sistema embebido forma parte de un sistema de control crítico. Tomando como ejemplo el sector de la automoción, desde el año 2000 al 2010 el valor del software embebido en el valor total del coche ha pasado del 2 % al 13 % [Cha09]. En este caso, además de la verificación de los requisitos funcionales, es necesario comprobar el correcto funcionamiento de los mecanismos de tolerancia a fallos que hayan sido introducidos para garantizar el correcto funcionamiento en circunstancias adversas. Al ser el software un elemento "blando", es el más fácil de moldear y cambiar, pero al mismo tiempo es la forma más sencilla de hacer concesiones e incorporar comportamientos no deseados en el producto final.

En este contexto donde el software juega un papel tan decisivo, las memorias no deberían "olvidar" los datos que tienen almacenados, pero lo hacen. Repentinamente un bit cambia de estado y el significado de la información almacenada cambia radicalmente siendo los efectos impredecibles. Tal vez, en la mayoría de los casos, estos efectos no vayan más allá de un pequeño trastorno para el usuario, que se resuelve con el mecanismo clásico

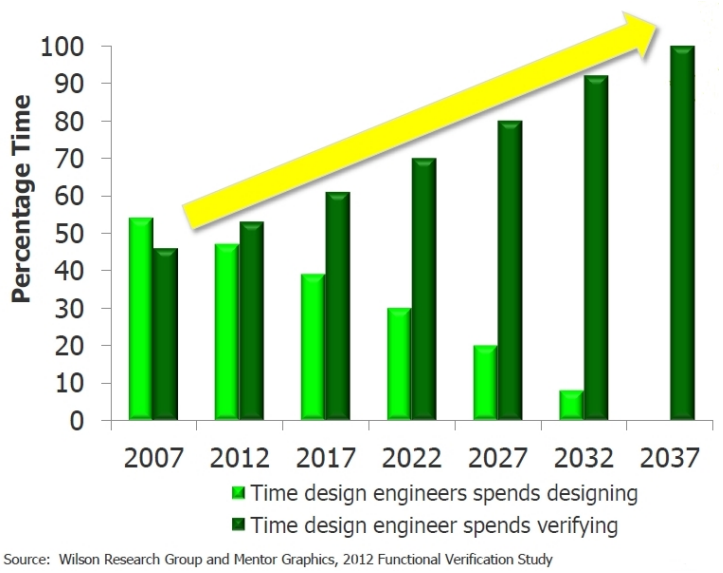


Figura 1.3: Crecimiento del esfuerzo dedicado a verificación

de *apagar y volver a arrancar* el sistema. Pero si el software formara parte de un sistema de control crítico, los efectos podrían ser más serios y causar pérdidas importantes, no solo de dinero o equipamiento, sino incluso de vidas.

El término *soft error* define, desde un punto de vista lógico, los efectos producidos por el cambio de estado transitorio de uno o más bits en un sistema electrónico debido a algún tipo de influencia exterior, como una interferencia electromagnética o el impacto de una partícula de alta energía, como un protón o una partícula alfa. De forma genérica se conoce como SEE [Kan11, Nic11, VM10], al suceso por el cual uno o más bits de un sistema digital cambian de estado de forma transitoria o incluso permanente. Este tipo de fenómenos fueron descritos por primera vez entre 1954 y 1957, al observarse anomalías en la instrumentación de medida en el transcurso de pruebas nucleares. El primer intento de explicación de estos sucesos fue postulado por Wallmark and Marcus en 1962 [WM62]. La primera descripción de efectos semejantes en un vehículo espacial fue descrita por Binder en 1975 [BSH75]. Otros trabajos pioneros como May and Woods [MW79] en 1979, comenzaron la investigación de *soft errors* en memorias dinámicas y los consiguientes efectos desde un punto de vista software. Una completa perspectiva histórica de los SEE desde el punto de vista de un fabricante de memorias puede encontrarse en [ZP04].

Hoy en día estos efectos no ocurren solo en entornos de alta radiación o ambientes espaciales. El gran público se ha visto afectado en acciones tan cotidianas hoy en día como los vuelos comerciales a gran altitud hasta los grandes centros de datos situados a nivel del mar [Ler07]. Desde los primeros sistemas integrados, la densidad de la circuitería electrónica no ha dejado de crecer. A ello se une la búsqueda de nuevas tecnologías para la reducción del consumo de los sistemas electrónicos alimentados con baterías. El efecto lateral no deseado ha sido un aumento en la susceptibilidad de estos sistemas al impacto

de partículas. En el año 2000 Sun Microsystems, reconoció problemas en las memorias cachés de sus servidores debido a rayos cósmicos que causaron problemas en servidores de empresas tan importantes como America Online o eBay [GGO03]. En 2008, el cambio de estado de un bit provocó la paralización del servicio durante varias horas por parte de los servidores de Amazon [Ser08]. Pero no solo los grandes sistemas de información han sufrido estos efectos, dispositivos tan sensibles para la vida de una persona como son los marcapasos electrónicos implantables, tampoco han sido ajenos a estos fenómenos [BN98, THMC12].

Estos efectos no deseados derivan en la necesidad de mejores y más eficientes metodologías de desarrollo y verificación de los sistemas, de forma que las técnicas de tolerancia a fallos implementadas en los diseños sean correctamente verificadas.

1.2.1. Desarrollo temprano de software depediente del hardware

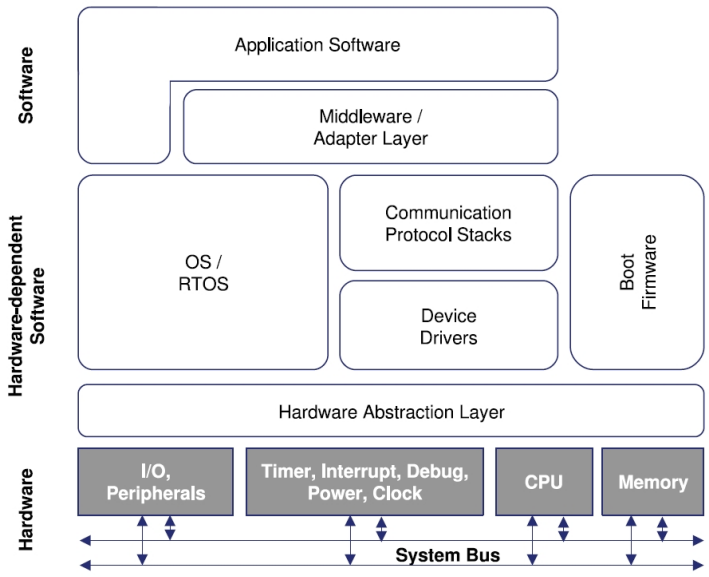


Figura 1.4: Software dependiente del Hardware [EMD09]

La figura 1.4 muestra un modelo conceptual con los principales componentes software que forman parte de un sistema embebido complejo. El número de componentes puede variar de un sistema a otro, eliminando alguno de ellos o incorporando otros nuevos, pero la figura representa una buena aproximación.

El código de aplicación, junto con las capas de adaptación (*middleware*) a los servicios del sistema operativo, ocupan la parte alta del diagrama y conforman el código que es independiente del hardware. De hecho, su interfaz es el sistema operativo. Por ejemplo, si un determinado sistema embebido va a incorporar Linux, el desarrollo de la aplicación puede ser realizado sin problemas en un sistema de sobremesa que disponga del mismo

sistema operativo, con la garantía de que, desde un punto de vista funcional, el código *debería* funcionar sin cambios en el sistema embebido final.

Los siguientes componentes incluyen código dependiente del hardware en diferentes proporciones. Los controladores de dispositivo (*Device Drivers*) proporcionan el código específico de manejo de un periférico en concreto, mientras el nivel de abstración hardware (HAL) proporciona los mecanismos básicos de acceso a los recursos hardware, como puede ser la instalación de manejadores de interrupción.

Mención especial merece el software de arranque (*boot*), el cual maneja el proceso de arranque del sistema, realiza una diagnosis básica y construye las estructuras de control primarias que necesita el procesador para su correcto funcionamiento, como es la tabla de tratamiento de interrupciones. Este módulo, además presenta una dificultad adicional ya que una parte sustancial del código ha de escribirse en el lenguaje de ensamble propio del procesador.

Todo el software dependiente del hardware, por definición, necesita de la existencia del hardware para su ejecución y prueba. Retrasar el comienzo del desarrollo de esta parte del software hasta la disponibilidad del hardware es una metodología que introduce retrasos no asumibles.

1.2.2. Prueba de los requisitos de tolerancia a fallos

Para garantizar el correcto funcionamiento de los mecanismos de excepción, es necesario ejercitarlos y ello no siempre es una tarea sencilla. En el diseño de la planta nuclear de Fukushima se estableció como requisito que los muros que la separaban del mar deberían ser capaces de soportar olas de hasta aproximadamente 6 metros [Com11]. Sin embargo el 11 de Marzo de 2011, una ola de 11 metros generada por un tsunami arrasó el complejo con graves consecuencias de contaminación radioactiva. De la misma forma, en el diseño software de sistemas críticos hay mecanismos de tolerancia a fallos que son avanzados en la especificación de requisitos del sistema. Estos mecanismos necesitan ser probados y acotada su validez dentro del entorno de actuación "esperado" del sistema. Aun así, tal y como sucedió con Fukushima, las previsiones pueden quedarse cortas y los efectos ser impredecibles.

Desde un punto de vista software, el análisis de comportamiento de un sistema en presencia de fallos puede revelar comportamientos no deseados, que no son descubiertos mediante los procedimientos normales de test. Primero, pone a prueba los mecanismos de excepción y fallo que en circunstancias normales no son suficientemente ejercitados y ayuda a evaluar el riesgo, verificando cuán malo puede llegar a ser el comportamiento de un sistema [VCM+97] en la presencia de fallos. Un sistema crítico debe proporcionar una respuesta adecuada, incluso cuando el uso es incorrecto o en condiciones de funcionamiento degradadas [CN13, VN56] y en el peor de los casos lo que no debe es empeorar la situación.

Como se desprende de la figura 1.5, la mayor parte del esfuerzo se dedica a la verificación de requisitos funcionales. La figura es una adaptación del modelo de distribución de fallos tomado de [Lyu96]. Muestra la densidad de fallos software no descubiertos a medida que avanza la fase de prueba. La figura también hace hincapié en el hecho que grandes cifras de cobertura no siempre significan una mejor verificación de los mecanismos de tolerancia a fallos. Típicamente, los mecanismos de excepción quedan fuera de los procedimientos clásicos de prueba, ya que los escenarios de prueba deben ser artificial-

mente generados y simplemente manejan ese tipo de situaciones que *"no pueden suceder si se sigue todas las reglas de un buen diseño e implementación software"*. Esto lleva a la paradoja de que los procedimientos que deben manejar situaciones excepcionales son raramente ejercitados y menos aún comprobados exhaustivamente. La inyección de fallos es una técnica de verificación experimental, que pretende verificar si los mecanismos de tolerancia a fallos se comportan tal y como fueron especificados y evaluar su cobertura, esto es, el porcentaje y la clase de fallos que pueden manejar. Todos los métodos de inyección de fallos están basados en características hardware/software concretas del sistema al que se aplica, por lo que la generalización es muy complicada. Una posible solución a estos problemas es el uso de plataformas virtuales de desarrollo.

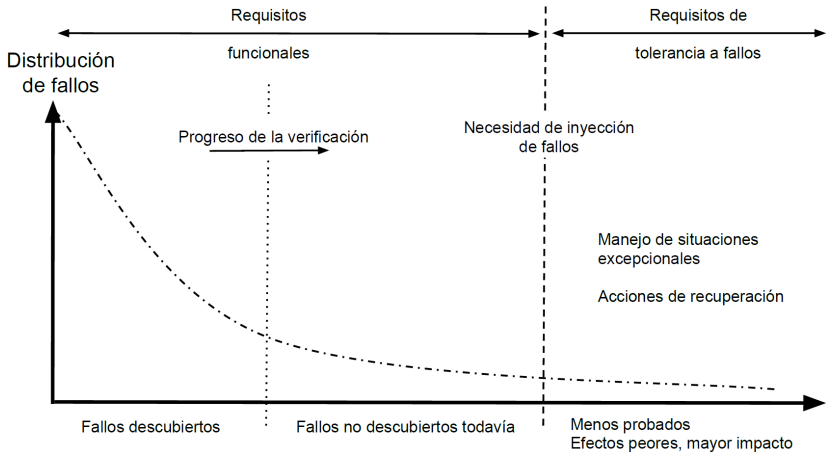


Figura 1.5: Necesidad de inyección de fallos. Adaptación tomada de [Lyu96]

1.3. Plataformas virtuales

Las plataformas virtuales son modelos ejecutables de los sistemas bajo desarrollo que proveen al desarrollador de software embebido de un entorno de ejecución mucho antes de que el hardware real esté disponible. De esta forma se habilita el desarrollo concurrente del hardware y del software, reduciendo de forma significativa el tiempo de integración. Entre las ventajas del uso de plataformas virtuales para el desarrollo y verificación software caben destacar:

- Elimina las dependencias del desarrollo de la disponibilidad del hardware.
- Proporciona ciclos de edición-compilación-prueba más ágiles en un entorno de ejecución más controlable, observable y determinista en la ejecución del software.
- Proporciona capacidades de depuración e inyección de fallos que no serían posibles de otra manera.

- Proporciona la capacidad de conectar sistemas reales mediante interfaces de comunicación estándar como líneas serie, ethernet o spacewire, lo que facilita la depuración y prueba del software embebido en comunicación con otros elementos del sistema, virtuales o reales, aproximación conocida como *Virtual Hardware in the Loop* (VHIL).

1.3.1. SystemC/TLM2

Los entornos de modelado de sistemas, permiten comenzar el proceso de diseño desde un punto de vista más abstracto y aplicar una metodología descendente *top-down*. El objetivo es relegar los detalles de implementación hasta el final con la intención de evitar que los *arboles impidan ver el bosque*. SystemC [OSC14] proporciona un mecanismo estándar, aceptado por la industria, para el modelado y verificación de conjuntos hardware/software usando una librería de objetos e interfaces C++. Estos modelos proporcionan una versión ejecutable de la especificación, lo que permite una simulación muy rápida y la verificación temprana de algunos requisitos del sistema. Además, las herramientas de desarrollo son compiladores, depuradores y entornos de desarrollo C++, ampliamente usadas y conocidas por los equipos de desarrollo, disminuyendo la brecha entre desarrolladores hardware y software.

El modelado en el nivel de transacción, TLM, eleva el nivel de abstracción de la descripción del comportamiento de un sistema, haciendo hincapié en el intercambio de datos entre los diferentes componentes, a través de canales de comunicación abstractos como *colas* de mensajes. El modelado TLM permite la exploración del espacio de diseño y la evaluación de diferentes alternativas así como la estimación de parámetros no funcionales esenciales en el desarrollo de sistemas embebidos, como son los tiempos de respuesta y el consumo energético.

La librería de interfaces SystemC/TLM2 es un estándar para el modelado de canales de comunicación basados en buses, como los que se pueden encontrar en cualquier sistema basado en procesador, memoria y periferia. Como punto de partida, el objetivo principal de TLM2 es facilitar el desarrollo temprado del software embebido, relacionando más estrechamente los flujos de diseño hardware y software, que tradicionalmente habían ido por separado.

1.4. Contexto de la tesis

Esta tesis ha sido realizada en el Grupo de Investigación del Espacio (SRG) de la Universidad de Alcalá en el marco del proyecto Solar Orbiter. Solar Orbiter [ESA11] es un satelite de observación solar actualmente en desarrollo por parte de la Agencia Espacial Europea (ESA) y cuyo lanzamiento está previsto para 2017. La misión de Solar Orbiter es obtener medidas detalladas del viento solar y realizar observaciones cerca de los polos solares. El grupo SRG está a cargo del diseño e implementación del conjunto hardware/software de la Unidad de Control del Instrumento (ICU) del Detector de Partículas Energéticas (EPD) embarcada en Solar Orbiter.

En su punto más próximo al Sol, alrededor de las 0,3 unidades astronómicas, véase la figura 1.6, la nave estará expuesta a una radiación solar del orden de 13 veces la radiación recibida en la tierra. Ello supone una alta probabilidad de SEEs, que deben

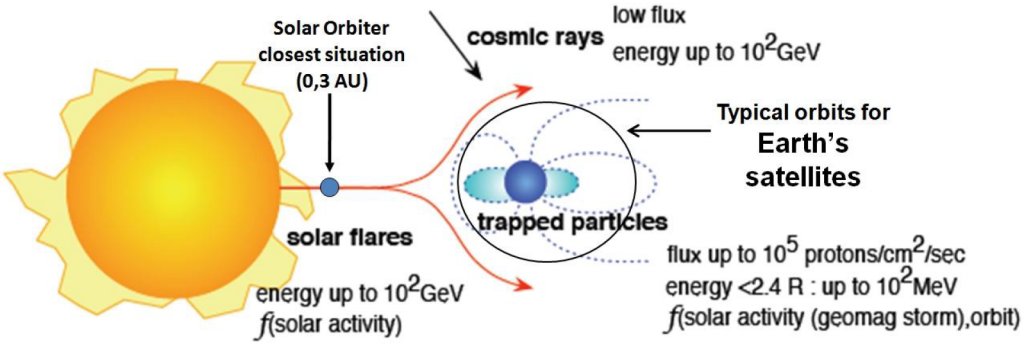


Figura 1.6: Posición relativa de Solar Orbiter respecto al Sol

ser adecuadamente tolerados por el sistema. Concretamente, el software de *boot* debe ser capaz de detectar la hipotética corrupción de los binarios de aplicación almacenados en EEPROM, realizar una verificación del estado de la memoria SDRAM y, en caso de ser necesario, realizar el grabado de una nueva versión del software de aplicación que evite las zonas dañadas. La verificación de los mecanismos y procedimientos de tolerancia a fallos especificados para el software es una tarea compleja, especialmente para el software de *boot*. Para ello se ha desarrollado una plataforma virtual "Leon2ViP", basada en interfaces SystemC/TLM2 con capacidad de inyección de fallos. De esta forma, ha sido posible ejecutar exactamente el mismo binario, pero en un entorno más controlado, permitiendo una estricta verificación de los procedimientos de arranque.

1.5. Estructura de la tesis

La presente memoria consta de cinco capítulos. El capítulo actual sirve de introducción y en él se reflejan de forma breve el contexto y los principales objetivos de investigación. En el capítulo dos, se presentan los conceptos fundamentales sobre los que se basa la investigación y se realiza una revisión de las técnicas y experiencias actuales en el diseño y verificación de software embebido en sistemas críticos en el ámbito espacial. En el capítulo tres se describe la plataforma virtual desarrollada en el marco del proyecto, con el fin de realizar una prueba exhaustiva del proceso de arranque del software de *boot*, las características que la hacen útil para el desarrollo temprano de software embebido y la posterior verificación de los requisitos de tolerancia a fallos que debe cumplir. El capítulo cuatro describe los resultados obtenidos en la verificación del software de *boot* mediante "Leon2ViP". El capítulo cinco contiene las conclusiones obtenidas como consecuencia del empleo de la plataforma virtual en el desarrollo real del software de *boot* para la ICU del instrumento EPD, así como las futuras líneas de investigación.

Por último, el anexo es un compendio de los principales artículos publicados en congresos internacionales y revistas que han sido generados fruto de la tesis: [dSS09b, dSS09c, dSS10, dSS11b, dSSM+10, dSS11a, SPP+13, dSSPP14, dSPPS].

Capítulo 2

Análisis del problema

En teoría, no existe diferencia alguna entre teoría y práctica; en la práctica sí la hay.
Jan L. A. van de Snepscheu

2.1. Introducción

Los sistemas embarcados para aplicaciones empleadas en aviónica o espaciales no han sido ajenos al espectacular crecimiento del software embebido [NAS01]. Inevitablemente, unido al crecimiento del volumen del software, ha crecido la cantidad de problemas o mal funcionamientos asociados al software de vuelo, como se puede apreciar en la figura 2.1.

Los problemas relacionados con el software embebido no siempre son debidos a errores de programación, sino en gran medida a su uso en un escenario para el que no había sido diseñado. En general, las causas de fallos pueden clasificarse de la siguiente manera:

- Ideas previas erróneas acerca de los requerimientos del sistema.
- Especificación incorrecta del sistema y/o del entorno de funcionamiento del mismo.
- Integración deficiente de componentes software de diversos proveedores o de libre distribución.
- Pruebas deficientes o incompletas de todos los escenarios en los que el software puede encontrarse. En este apartado pueden incluirse todas las situaciones en las que los valores almacenados en memoria se ven alterados por alguna influencia externa.

Un estudio de *ElectricCloud* [Clo10] revela que la mayoría de los *bugs* son debidos a procedimientos de test pobres o inadecuados, a causa de las limitaciones de la organización y de las herramientas, más que a problemas de diseño. De forma adicional, la fase de prueba es generalmente considerada como un proceso profesionalmente poco placentero. Ello deriva en la necesidad de nuevas técnicas, herramientas y metodologías.

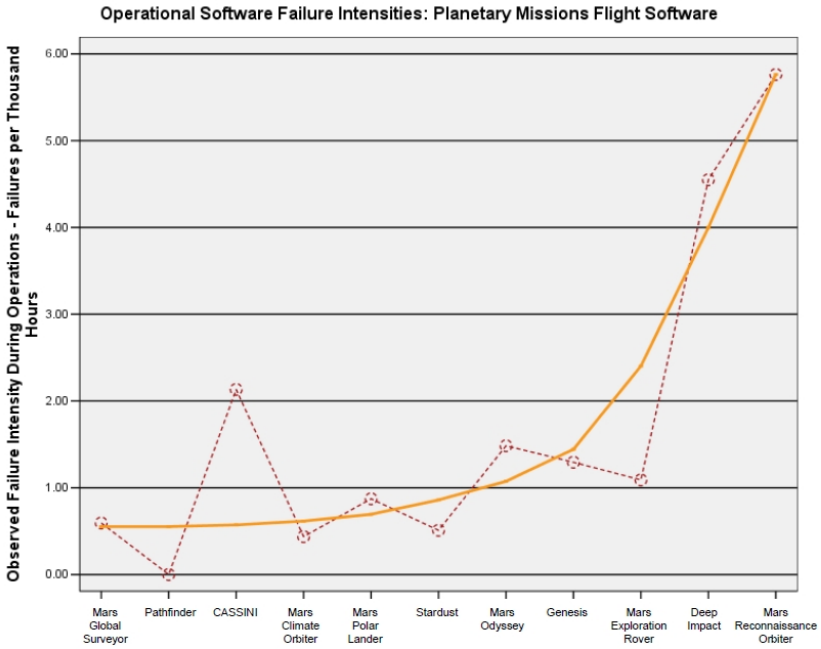


Figura 2.1: Incremento de los fallos del software de vuelo [NTC11]

2.2. Efectos de la radiación sobre las memorias

Las memorias electrónicas que operan en el espacio tienen una alta probabilidad de perturbación, principalmente debido al impacto de partículas con carga, electrones y protones sueltos y partículas alfa, efecto conocido como SEEs. Otra posible fuente de perturbación es la absorción acumulada de radiación, conocida como *Total Ionizing Dose* (TID), la cual provoca una degradación de los contenidos a lo largo del tiempo. El proceso físico por el cual se produce esta modificación en la circuitería digital en general, y en las memorias en particular, está fuera de la línea de investigación de esta tesis. El interés de la tesis está en la verificación software de los mecanismos de recuperación, una vez que las alteraciones se han producido.

La figura 2.2 muestra la tendencia en el incremento de SEEs. Estos efectos se han incrementado sustancialmente debido a dos aspectos relacionados con el aumento de la escala de integración y la reducción del consumo.

- Aumento de la escala de integración. El incremento del software va inevitablemente unido al incremento de la capacidad de almacenamiento de las memorias. Una mayor escala de integración implica celdas de memoria más reducidas y más susceptibles a la modificación debido al impacto de alguna partícula. Otro efecto lateral es el incremento en la alteración simultánea de varias celdas.

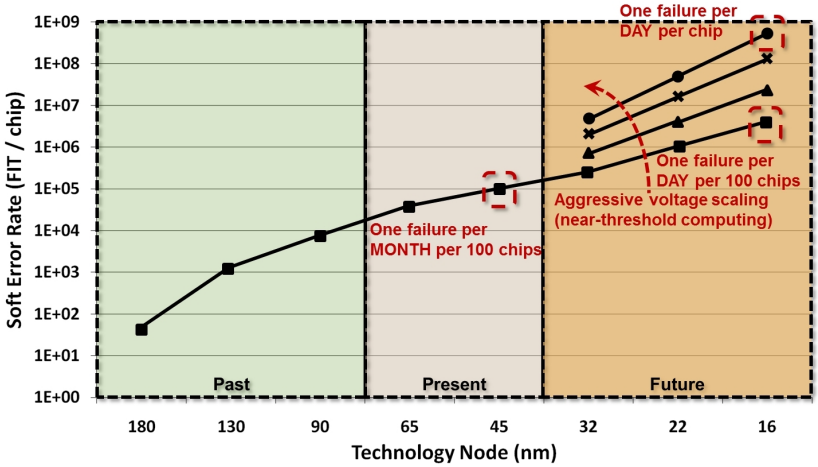


Figura 2.2: Tendencia SEE [SKK+02]

- Reducción del consumo. Consumir menos significa usar menos energía para almacenar el estado de una celda de memoria. De nuevo la susceptibilidad de modificación ante un aporte de energía externo se ve incrementada.

En esta tesis se usa el termino *soft error* para referirse a los efectos, desde un punto de vista software, del impacto de partículas que provocan algún tipo de alteración de los valores almacenados en las memorias. Estos efectos, dependiendo de cómo el impacto afecta a la circuitería, pueden ser clasificados en los siguientes tipos: [VM10, Nic11, KISU10].

- No destructivos (*Single Event Upset* (SEU)). Estos son efectos transitorios que alteran el valor almacenado en una celda de memoria. El fallo puede ser detectado y el valor correcto restaurado. La alteración de más de una celda se conocen como *Multiple Event Upset* (MEU).

- Destructivos (*Single Event Latchup* (SEL)). Estos efectos producen un daño permanente en una o más celdas de memoria. La memoria pierde la capacidad de almacenar diferentes valores y el valor almacenado permanece fijo.

2.2.1. Técnicas de mitigación

Partiendo del hecho de que los *soft error* han llegado para quedarse, es necesario aplicar diferentes técnicas de mitigación que permitan a los sistemas tolerar la presencia de estas incidencias y seguir prestando el servicio que tienen encomendado. En el peor de los casos, el error debe ser detectado y los posibles efectos negativos controlados.

Las diversas técnicas de mitigación, véase la figura 2.3, se sitúan en diferentes planos del diseño de un sistema y en algunos casos están relacionadas entre sí. De forma genérica pueden clasificarse en: [Dub13, SWI06]

Figura 2.3: Técnicas de mitigación SEEs

- Físicas: están relacionadas con los aspectos de construcción física del chip y la ubicación de las diferentes celdas de memoria. La idea principal es conseguir que celdas de memoria que son físicamente contiguas pertenezcan, desde un punto de vista lógico, a diferentes palabras de memoria. De esta forma, si se produce una alteración en más de una celda, habrá varias palabras de memoria con un error en vez de una única con varios errores. Esto enlaza con las técnicas de mitigación lógicas, en las que se protege cada palabra de memoria mediante códigos correctores de error.

- Lógicas: cada palabra de memoria es protegida mediante códigos *Error Detection and Correction* (EDAC) que introducen redundancia en los contenidos de las palabras de memoria. Este mecanismo garantiza que un fallo no destructivo en un bit puede ser detectado y corregido al realizar una operación de lectura de la palabra de memoria. Si hubiera más de un fallo se garantiza la detección del error pero no su corrección. Al software se le notificaría esta circunstancia mediante una excepción. Si la operación fuera de escritura se almacenaría el nuevo valor sin más. Puesto que el contenido de las palabras de memoria solo se comprueba y corrige al realizar un acceso, es imperativo realizar un refresco cíclico de la información almacenada antes de la llegada de un segundo impacto que pueda provocar un nuevo error, situación en la que el contenido de la palabra de memoria será irrecuperable. Este refresco cíclico es conocido como *Memory Scrubbing* [MRB+11, RMB11].

- Sistema: desde un punto de vista hardware es necesario conocer si el software está en un estado operativo. Los sistemas tipo *watch-dog* provocan un *reset* del sistema en caso de no ser reinicializados de forma cíclica. Además, el hardware crítico es protegido mediante redundancia hardware. La forma de redundancia más típica es conocida como *Triple Modular Redundancy* (TMR) [RMR11]. En ella los bloques funcionales protegidos se triplican y realizan la misma computación en paralelo, resolviéndose las discrepancias mediante un votador.

- Algorítmicas: de la misma forma que hay redundancias en la información almacenada en memoria y en los elementos funcionales del hardware, el software también puede disponer de redundancia en los datos y en los algoritmos de procesamiento para garantizar que en caso de corrupción irrecuperable de una variable, ésta pueda ser recuperada. Otros aspectos algorítmicos están relacionados con los mecanismos de recuperación del sistema una vez que todas las técnicas de mitigación han sido superadas. Por ejemplo, en caso de daños permanentes en ciertas zonas de memoria sería posible reconfigurar el software para que evite esas zonas.

2.2.2. Modelo de fallos en memoria

La alteración en el comportamiento de una celda de memoria es vista desde el software como la modificación de un bit. Los SEUs provocan cambios de estado transitorios de los bits conocidos en la literatura como *bit-flips*. Los SELs inhabilitan al bit para cambiar de estado, independientemente de las operaciones de escritura que se hagan sobre él. Estos bits son conocidos como *stuck-at zero* o *stuck-at one* bits, en función de que su valor quede fijado a cero o a uno.

La modificación de uno o más bits de la memoria de un sistema procesador tiene diferentes efectos sobre el software, dependiendo de la zona del programa donde tenga lugar. De hecho, puede haber partes de la memoria sin usar, por lo que un error en esas localizaciones no tendría efecto alguno. El binario de un programa está generalmente formado por diferentes secciones y un error en alguna de ellas afecta de forma muy diferente al conjunto. Las secciones más típicas que prácticamente todo programa contiene son:

- BSS: contiene todas las variables globales que son inicializadas por defecto a cero.
- DATA: contiene todas las variables globales que no son inicializadas por defecto a cero.
- HEAP: resto de la memoria dedicada a datos temporales, como puede ser la pila del sistema.
- TEXT: contiene el código de aplicación propiamente dicho.

El incorrecto funcionamiento del software embebido puede clasificarse de acuerdo al método *Failure Mode and Effects Analysis* (FMEA). FMEA es básicamente un análisis cualitativo del funcionamiento esperado del sistema. Dependiendo de la zona de memoria afectada, los comportamientos anómalos pueden ser clasificados como:

- Resultados incorrectos, temporización correcta.
- Resultados correctos, temporización incorrecta.
- Resultados incorrectos, temporización incorrecta.
- Bucle sin fin del procesador.
- Lecturas/escrituras no alineadas en memoria.
- Códigos de operación no válidos.

Para realizar una campaña de inyección eficiente y representativa, es conveniente realizar un estudio previo del mapa de memoria de la aplicación [GKS+12, BVFK05].

2.3. Inyección de fallos

Una parte importante del código en un sistema crítico tiene como misión tratar situaciones excepcionales y proporcionar los mecanismos básicos de recuperación del sistema o su parada controlada. La inyección de fallos es una técnica útil, cuando no imprescindible, para la verificación de los mecanismos de excepción concretos implementados en un sistema.

2.3.1. Taxonomía de los fallos

Incluso en el sistema más sencillo existen una gran cantidad de puntos de inyección de fallos, [dSS11a]. El desafío es lograr una campaña de inyección representativa del fallo que se desea provocar y del mecanismo de excepción que se pretende probar. Las cuestiones fundamentales se resumen en lo que se conoce como el dilema W.W.W. (*Where, When, What*):

- ¿Dónde inyectar? Desde un punto de vista software es imprescindible un conocimiento preciso del mapa de memoria. Dependiendo del uso concreto de cada localización de memoria, el efecto sobre el sistema puede ser muy diferente. Por ejemplo no tendría sentido inyectar fallos en zonas no usadas.
- ¿Cuándo inyectar? En esta pregunta se resumen todas las condiciones que deberían cumplirse para provocar el "disparo" de un fallo, conocidos en la literatura como *triggers*. Durante la ejecución del código de una aplicación las variables presentan periodos de actividad en los cuales almacenan información "viva". Durante otros periodos, los valores almacenados son intrascendentes y serán sobreescritos por nuevos valores. Si se pretende evaluar la susceptibilidad al error de una, o un conjunto de variables, es preciso inyectar fallos en instantes bien definidos o bajo condiciones de ejecución del programa muy precisas. Un buen abanico de condiciones de disparo de los fallos mejora la reproducibilidad de los experimentos y la obtención de conclusiones fiables acerca del comportamiento del sistema.
- ¿Qué tipo de corrupción debe ser aplicada y cúanto tiempo dura el fallo? Los patrones de corrupción típicos son los *bit-flips* y los *stuck-at*. Los *bit-flip* simulan cambios transitorios y los *stuck-at* permiten simular fallos intermitentes y/o permanentes.

2.3.2. Técnicas de inyección de fallos

Establecido el tipo de fallo que se pretende provocar es preciso encontrar la técnica que permita realizar la inyección. Aparte de los atributos específicos de la inyección, otros aspectos importantes son la controlabilidad, observabilidad y reproducibilidad de los experimentos:

- Controlabilidad: es la capacidad del procedimiento de inyección de "tocar" el sistema de forma que se sitúe en el estado deseado.

- Observabilidad: es la capacidad del procedimiento de inyección de "ver" el estado interno del sistema y seguir la propagación de un fallo.
- Reproducibilidad: es la capacidad del procedimiento de inyección de "repetir" un determinado escenario de prueba de forma que se puedan obtener resultados estadísticos fiables.

Es importante señalar que la descripción de las ventajas e inconvenientes de cada técnica se realiza desde el punto de vista de la verificación de los mecanismos de tolerancia a fallos del software embebido, objetivo principal de esta tesis.

2.3.2.1. Inyección de fallos por hardware

Se define la inyección de fallos por hardware como aquella que usa circuitería o elementos físicos adicionales para alterar el estado de los bits del sistema. La inyección puede ser *con contacto*. En este caso se accede directamente mediante el uso de sondas u otros elementos a los pines o puntos de prueba presentes en el sistema final. Puesto que no es posible disponer físicamente de todos los posibles puntos de prueba y error, éstos deben ser cuidadosamente elegidos durante el diseño y fabricación de la placa [ZWGC10]. En los métodos *sin contacto* se provocan los fallos mediante la exposición del sistema bajo prueba a un haz de radiación, rayos laser o interferencias electromagnéticas. Los métodos de inyección de fallos sin contacto son los procedimientos más realistas ya que reproducen la naturaleza real del fallo. Por contra son los menos controlables y reproducibles, al ser imposible fijar el instante y localización de la inyección. Como desventaja adicional, estos métodos presuponen la existencia real de un hardware que probar por lo que se encuadrarían en un ciclo de diseño *software-after-hardware* clásico.

2.3.2.2. Mecanismos de depuración propios del hardware

En este apartado se encuentran las interfaces de depuración propias incluidas en algunos procesadores y/o microcontroladores conocidas de forma genérica como *On-Chip Debugger* (OCD), por ejemplo *Joint Test Action Group* (JTAG) [IEEE Std 1149.1 01], Nexus [IEEE-ISTO 5001 03] o *Background Debug Mode* (BDM). JTAG y Nexus son interfaces estandarizadas y proporcionadas por diferentes fabricante, BDM es una solución propietaria desarrollada por Freescale. Aunque estos sistemas permiten una mejor localización de la inyección, tienen el inconveniente de ser bastante intrusivos desde el punto de vista de la ejecución del software. Por ejemplo, para acceder vía JTAG al estado interno de un sistema, éste debe ser típicamente detenido (*freezed*), siendo impracticable la inyección de fallos de forma permanente. De nuevo el hardware debe existir previo al software, lo que hace imposible un desarrollo temprano del software.

2.3.2.3. Inyección de fallos por software

La inyección de fallos por software, *Software Implemented Fault Injection* (SWIFI), resuelve algunas de las cuestiones planteadas previamente. Mediante código específico adicional es posible acceder y modificar cualquier elemento lógico accesible al software. La localización de los fallos y la reproducibilidad de las campañas de inyección es superior a las técnicas anteriores, siendo posible elegir prácticamente cualquier configuración

software como condición de disparo del fallo. Sin embargo, no es posible emular fallos permanentes en memoria y además no cumple el paradigma *test what you fly, fly what you test* ya que el software es modificado para realizar la inyección. La existencia previa del hardware enmarca la técnica en una metodología de verificación software clásica. Un ejemplo de herramienta SWIFI y su adaptación a diferentes entornos de ejecución puede encontrarse en [dSML+07, dSGCM+09]. La inyección de fallos en modelos hardware VHDL mediante software de emulación podría encajarse en este o en el siguiente apartado 2.3.2.4. Un ejemplo de herramienta de inyección en cores o circuitos de procesamiento digital se encuentra descrito en [RMRR07].

2.3.2.4. Inyección de fallos en modelos o plataformas virtuales

La inyección de fallos en modelos virtuales elimina los problemas expuestos previamente. Por supuesto, los resultados obtenidos son completamente dependientes de cuán bueno es el modelo para el requisito cuya verificación concreta se desee realizar. En primer lugar, la verificación de ciertos aspectos del software puede comenzar tiempo antes de la existencia del hardware. La inyección de fallos es no intrusiva, controlable y debido a la naturaleza virtual del modelo, la propagación de los efectos del fallo son perfectamente observables y reproducibles. Otro punto importante que hay que tener en cuenta es que, al ser la plataforma de ejecución virtual, es posible ejecutar simultáneamente varios casos en paralelo, elimnando la dependencia de la disponibilidad del hardware y reducciendo el tiempo total dedicado a la campaña de fallos.

2.4. Modelado de sistemas en el nivel de transacción

La ingeniería dirigida por modelos [Gro] ataca la dificultad del diseño de sistemas complejos mediante el incremento de los niveles de detalle de un modelo. Se parte de modelos abstractos e independientes de la plataforma de ejecución *Platform Independent Model* (PIM) que de forma sistemática se van refinando hasta definir los modelos específicos de la plataforma *Platform Specific Model* (PSM). El modelado de sistemas en el nivel de transacción (TLM) podría definirse como [Ghe06, Gro02, CDBA10]:

> TLM es una aproximación al diseño de sistemas digitales donde los detalles de la comunicación entre los diferentes módulos se separan de los detalles de implementación de las unidades funcionales. La idea subyacente es modelar únicamente el nivel de detalle necesario en cada etapa del diseño para el desarrollo de una tarea o unidad funcional en particular.

Mediante una descripción TLM se puede validar el comportamiento global de un sistema sin entrar necesariamente en detalles de implementación. TLM es una metodología utilizada en el proceso de diseño y verificación que permite:

- Verificar y explorar alternativas en el diseño de sistemas.
- Particionar e integrar modelos HW y SW mediante sucesivas etapas de refinamiento.
- Verificar de forma funcional el HW en etapas iniciales. TLM proporciona un modelo de referencia, *Golden Model*, del hardware.

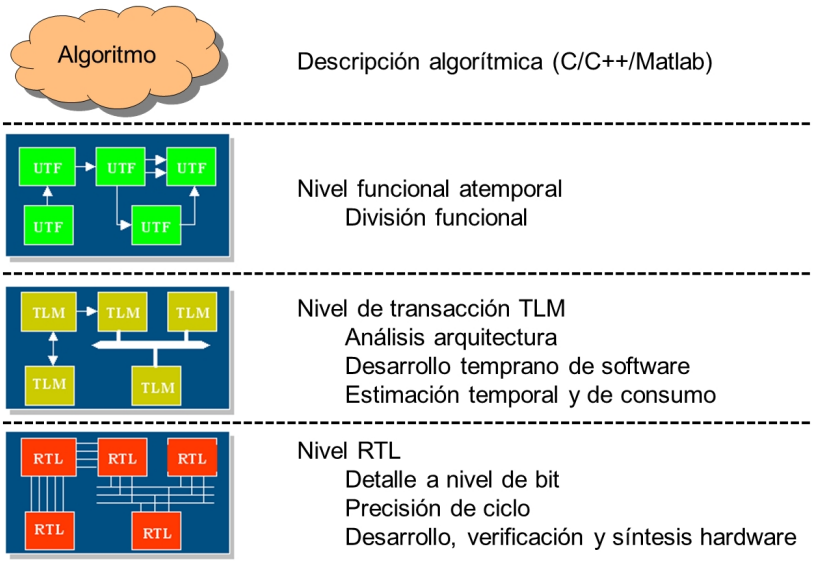


Figura 2.4: Metodología TLM

- Proporcionar una plataforma de ejecución que permita comenzar con el desarrollo temprano del SW eliminando la dependencia entre las tareas relacionadas con el desarrollo SW y la disponibilidad real del HW.

La figura 2.4 muestra los diferentes niveles de refinamiento que, desde un punto de vista general, puede sufrir la especificación de funcionamiento de un sistema. Desde la visión más abstracta en forma de algoritmo hasta el nivel *Register Transfer Level* (RTL) que ya es directamente sintetizable en hardware.

2.4.1. Estilos de codificación

Dependiendo del nivel de precisión en el modelado de un componente, el *Open SystemC Initiative* (OSCI) TLM WorkGroup [OSC14] define varios estilos de programación:

Atemporal - *Untimed* En este tipo de modelado no hay mención explícita al tiempo o ciclos que una operación puede durar, pero se respeta la secuencia de eventos del sistema.

Temporal poco precisa - *Loosely Timed* El modelado *temporal poco preciso* incluye la información temporal mínima para arrancar un sistema operativo y para manejar varios hilos de ejecución sin sincronización explícita entre ellos. Para ello, el modelo debe incluir un temporizador básico que notifique el transcurso de una ranura de tiempo. Esta ranura de tiempo simulado podrá coincidir con el tiempo real de simulación. Este tipo de modelado suele implementarse mediante una API bloqueante en el sentido de que cada transacción se completa mediante la invocación de una rutina. Desde un punto de vista

software, este nivel de modelado es conocido como `Visión del programador (PV)` ya que permite el soporte mínimo para el *boot* de un sistema y la verificación funcional del software.

Temporal aproximada - *Approximately-Timed* En este modelo se detallan de forma aproximada el protocolo de acceso y el comportamiento interno de un componente. Por ejemplo, en el caso del acceso a una memoria se detallarían las fases de direccionamiento, la posible latencia en la respuesta y el acceso a los datos. Lo que está modelado de forma aproximada es la secuencia de operaciones pero no la precisión de cada una de ellas. Este tipo de modelado suele implementarse mediante una API no bloqueante en el sentido de que cada transacción se completa mediante la invocación de varias rutinas que modelan las diferentes fases del intercambio de la información de la transacción. Desde un punto de vista software este estilo de codificación es denominado como `Visión del programador con temporización (PVT)`. En este estilo de modelado se mejora la estimación de los aspectos temporales y la verificación de requisitos no funcionales relacionados con las resticciones temporales.

2.4.2. Niveles de abstracción

De acuerdo con [BD86], *esencialmente todos los modelos son erróneos pero algunos son útiles*. En el modelado de un sistema existe siempre un balance entre el nivel de precisión del modelo y la velocidad de simulación del mismo. A la hora de modelar un componente hay que distinguir dos aspectos fundamentales: la precisión en el comportamiento interno y la precisión en comunicación con los demás componentes. El modelado en el nivel de transacción aumenta el nivel de abstracción en la comunicación, centrándose únicamente en la información intercambiada, independientemente del mecanismo hardware usado para dicha comunicación.

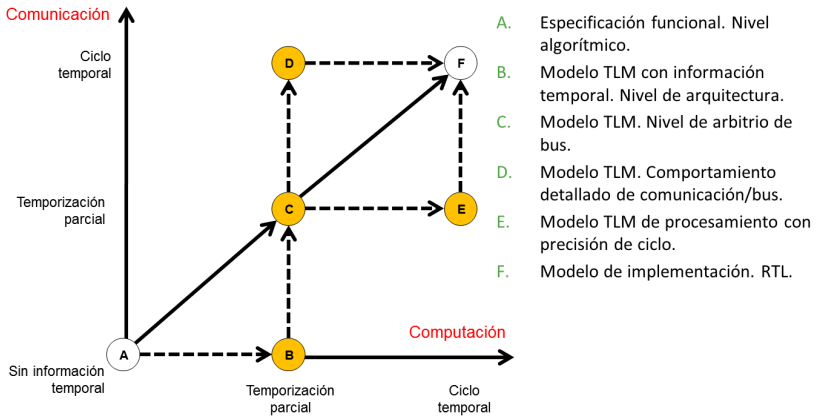


Figura 2.5: Niveles de abstracción y refinamiento. Adaptación tomada de [CG03]

La figura 2.5 muestra los diferentes niveles de abstracción existentes atendiendo a los dos aspectos comentados previamente: detalle o precisión del modelado del comportamiento interno y de la comunicación con el resto del sistema y las sucesivas etapas de refinamiento que pueden sufrir los modelos siguiendo la metodología TLM.

2.4.3. Flujo de diseño TLM

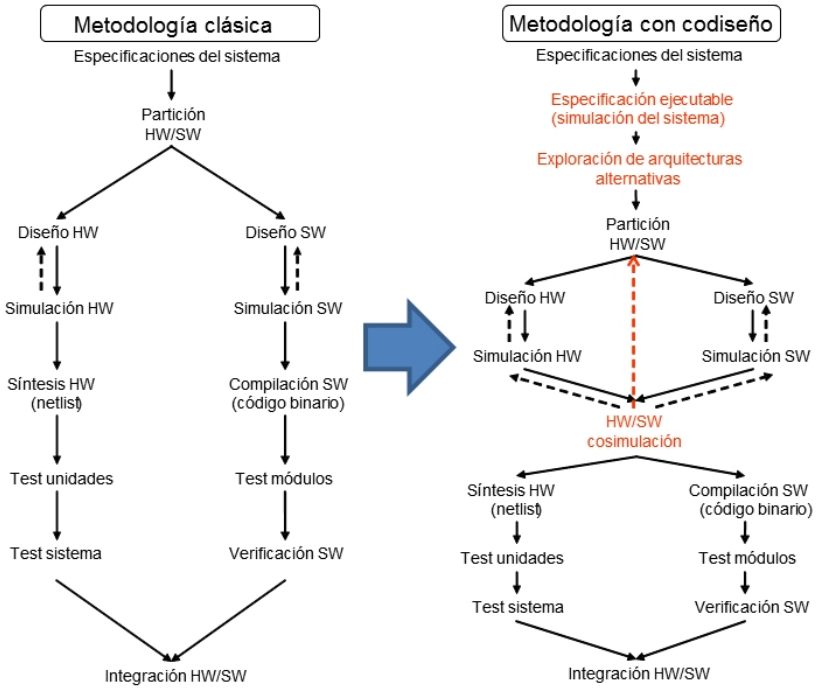


Figura 2.6: Flujo de desarrollo en TLM

El diseño de sistemas embebidos ha impulsado la integración de los equipos de diseño electrónico y de desarrollo de software, equipos y flujos de desarrollo tradicionalmente separados [Ghe06, BBM+10]. Así, se impulsa la propuesta de flujos de codiseño HW/SW, véase la figura 2.6. Tradicionalmente los flujos de diseño de sistemas embebidos han tenido forma de "Y", en los que después de un particionado temprano del sistema, los flujos de diseño HW y SW arrancan por separado, para confluir posteriormente en una parte característica y compleja donde se realiza la integración HW/SW. En estos flujos, se introduce una etapa inicial y común consistente en la especificación del sistema. En algunos casos, esta especificación se reduce a un grupo de documentos cuya semántica y requisitos son ambiguos. Esto dificulta la cooperación en el grupo de diseño, así como la compatibilidad de las implementaciones cuando se reutiliza la especificación. Este modelo no es lo suficientemente productivo porque usualmente precisa secuenciar estos flujos, siendo necesario comenzar el diseño de la plataforma HW antes. Además los problemas

relacionados con la especificación o la propia integración HW/SW se detectan muy tarde, lo que aumenta su coste en términos de dinero y tiempo de mercado.

La metodología TLM exige la puesta en común de los equipos de diseño HW y SW y tiene la forma de una "Y" invertida, elevando el nivel de abstracción en las etapas iniciales del diseño, para continuar por separado una vez decidido el particionado HW/SW del sistema en base a exploraciones del espacio de diseño. El uso de un modelo funcional, bien usando un lenguaje de programación de alto nivel (por ejemplo C) o bien un lenguaje de modelado (Matlab, SDL, UML, etc.), contribuye a restar ambigüedades. Mediante este modelo ejecutable es posible aplicar diferentes técnicas de exploración que van desde estratégias heurísticas basadas en el *know-how* del equipo de desarrollo hasta el uso de algorítmos bioinspirados [LGdVH13]. Una vez decidido el modelo TLM éste actúa como referencia en las etapas sucesivas del diseño. En estos flujos las especificaciones se ciñen a ser un punto de arranque del proceso de diseño. Se requiere aún el refinamiento de las interfaces HW/SW y traslaciones manuales: la partición HW a lenguaje HDL y la partición SW a un lenguaje de programación.

La figura 2.7 muestra las etapas y estilos de codificación existentes en una metodología de desarrollo TLM en la que se parte de un modelo más abstracto, que es sucesivamente refinado hasta el modelo directamente sintetizable.

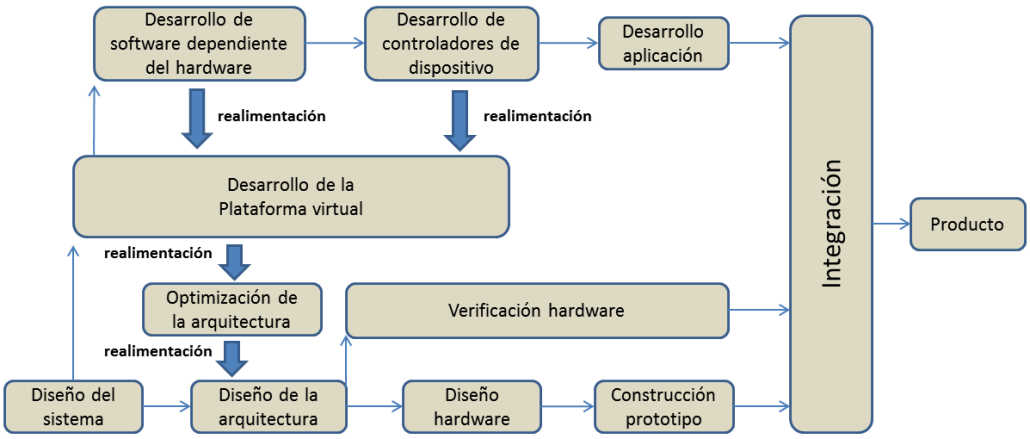


Figura 2.7: Flujo de desarrollo TLM con plataforma virtuales

2.4.4. Modelado SystemC/TLM

Aunque TLM es independiente del lenguaje de programación, es de mucha ayuda el hecho de que exista un lenguaje que lo soporte de forma que las descripciones puedan ser interoperables. SystemC ha sido el lenguaje precursor por excelencia del modelado TLM. Desde la primera versión (TLM 1.0) hasta el más reciente estándar OSCI TLM 2.0, se han presentado un conjunto de interfaces, que el diseñador de sistemas puede usar para intercomunicar los diferentes componentes de alto nivel del sistema bajo desarrollo. TLM 1.0 introdujo las interfaces unidireccionales:

- Bloqueantes: `tlm_blocking_get_if<T>`, `tlm_blocking_peek_if<T>`, `tlm_blocking_put_if<T>`
- No bloqueantes: `tlm_nonblocking_get_if<T>`, `tlm_nonblocking_peek_if<T>`, `tlm_nonblocking_put_if<T>`)
- Petición / respuesta: `tlm_transport_if<REQ,RSP>`, `tlm_master_if<REQ,RSP>`, `tlm_slave_if<REQ,RSP>`)

TLM 1.0 aportó también algunos elementos, como los canales `tlm_req_rsp_channel`, `tlm_transport_channel` y `tlm_fifo`, que enriquecen la capacidad de elección en semánticas de comunicación. Sin embargo, todas las interfaces descritas dejan abierta la definición de los datos intercambiados entre los componentes, lo que deriva en una gran diversidad de implementaciones de las interfaces que dificulta, cuando no impide, la interoperabilidad entre componentes proporcionados por diferentes fuentes.

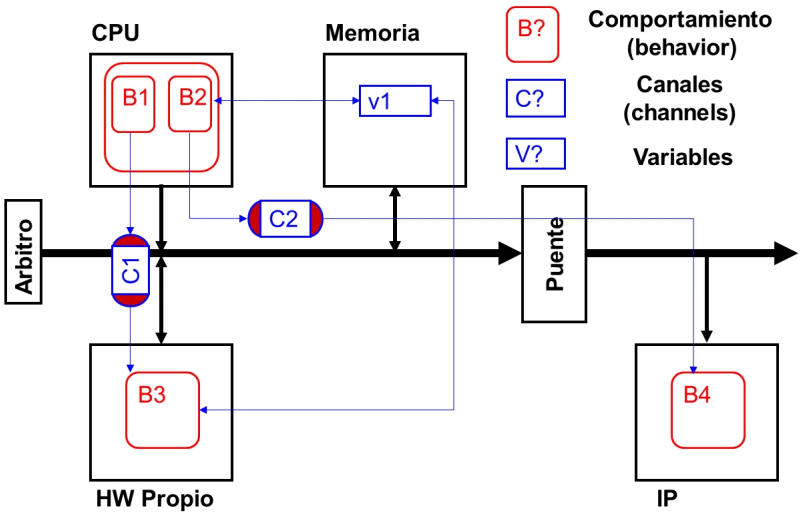


Figura 2.8: Modelado de una micro-arquitectura

Por ejemplo, mediante elementos de comunicación abstractos es posible describir una micro-arquitectura como la mostrada en la figura 2.8. El comportamiento de cada uno de los elementos activos de bus puede ser definido mediante su propio hilo de ejecución, usando una interfaz estándar de programación como puede ser *Portable Operating System Interface* (POSIX). Estos componentes se ejecutan de forma concurrente y se comunican mediante los canales de comunicación descritos previamente. El problema básico en este tipo modelado es que la información intercambiada no está estandarizada por lo que la interoperabilidad entre módulos descritos por diferentes proveedores no está garantizada.

2.4.4.1. Estándar TLM2.0

TLM 2.0 ha estandarizado los elementos necesarios para la comunicación. Se distingue entre módulos y conectores iniciadores de una transacción (maestros de bus) conocidos como *Initiator* y los objetivos (esclavos de bus) conocidos como *Target*. TLM 2.0 también define una carga útil genérica, que soporta el modelado abstracto de buses proyectados en memoria. En TLM 2.0 la interoperabilidad entre módulos es un objetivo clave y facilita el desarrollo de herramientas integradoras de componentes procedentes de distintas fuentes.

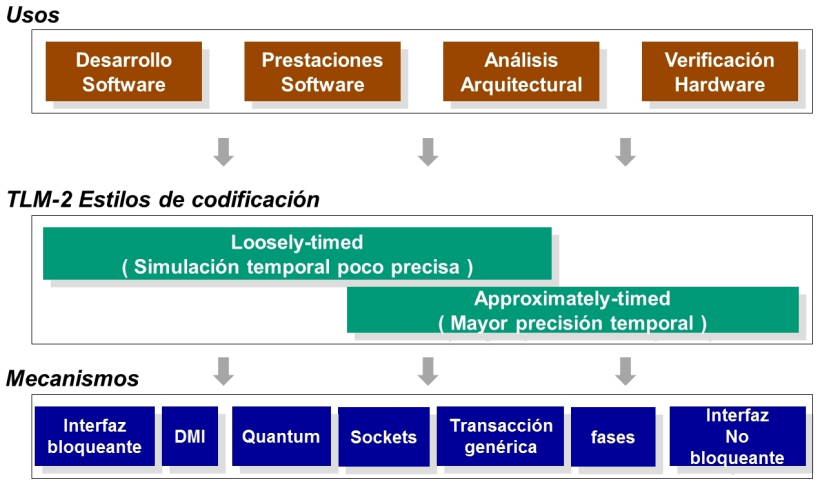


Figura 2.9: Usos, estilos y mecanismos en SystemC/TLM2

TLM2 define un conjunto de estilos de programación que ayudan a definir el alcance de los modelos, véase la figura 2.9. Estos modelos de codificación están basados en dos interfaces denominadas bloqueante y no bloqueante. Las interfaces están orientadas al modelado de las comunicaciones de sistemas basados en buses. Las primitivas básicas ofrecen servicios como la lectura o escritura de conjuntos de ráfagas de bytes a partir de una dirección. Entre otras cosas, al definir las primitivas de comunicación y el tipo de información intercambiada, facilita la interoperabilidad de los modelos escritos por diferentes proveedores y su integración en herramientas automáticas.

La idea principal consiste en comunicar la información estrictamente necesaria para simular la operación sin necesidad de simular todo el comportamiento hardware subyacente. En aras de la eficiencia, cada transacción se encapsula como parámetro en una única invocación, como se muestra en la figura 2.10. A las interfaces TLM 1.0 descritas previamente, TLM 2.0 ha añadido las siguientes:

- Bloqueante: `tlm_blocking_transport_if<TRANS>` En la interfaz bloqueante, todas las comunicaciones entre dos módulos se realiza mediante la invocación de una sola llamada (`b_transport()`). Esté método es invocado en el lado del iniciador y la respuesta implementada en la parte del módulo destino (*target*). Este mecanismo

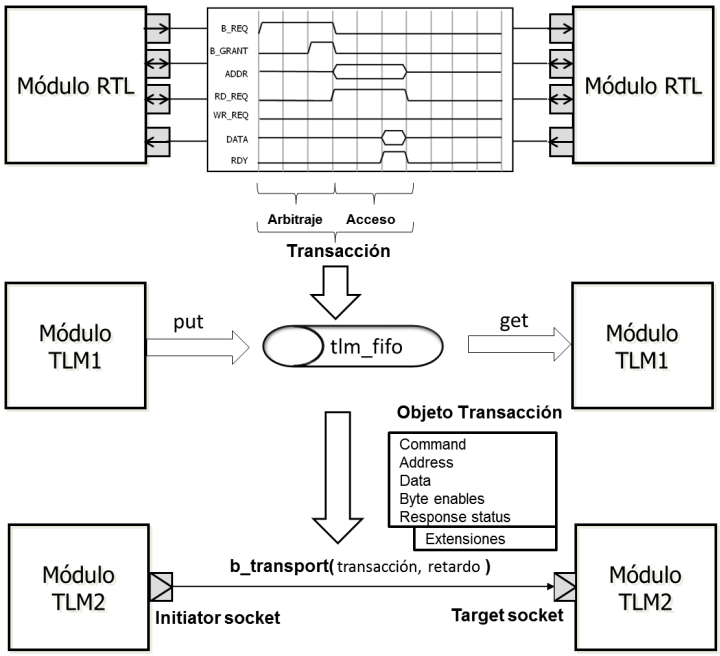


Figura 2.10: Transacciones en SystemC/TLM

tiene dos puntos de sincronización explícitos, en la llamada y en el retorno del método. La transacción se da por finalizada con el retorno del método.

- No bloqueante: `tlm_fw_nonblocking_transport_if<TRANS>` y `tlm_bw_nonblocking_transport_if<TRANS>` En esta interfaz, la comunicación entre dos módulos se realiza de una forma asíncrona usando dos llamadas, una para cada elemento de la comunicación. De esta forma, el iniciador invocará `nb_transport_fw` y la función retornará inmediatamente sin que la transacción necesariamente se haya completado. El módulo *target* invocará `nb_transport_bw` para completar la transacción. Este mecanismo proporciona cuatro puntos de sincronización.

- Interfaz de depuración: `tlm_trasport_dbg_if<TRANS>` La interfaz de depuración está pensada para proporcionar un acceso a la estructura interna del componente. La información intercambiada tiene el mismo formato de transacción.

- Interfaz de acceso directo a memoria: `tlm_dmi_if<TRANS>` La interfaz *Direct Memory Interface* (DMI) proporciona un acceso directo a los datos del *target* de forma que la velocidad de acceso se acelera. El iniciador solicita el acceso y el *target* podrá concederlo o no. Además el *target* puede en cualquier momento invalidar el acceso.

2.4.5. Modelado del procesador

Desde el punto de vista del desarrollo de software dependiente del hardware, el modelado de los procesadores es fundamental para poder realizar la cosimulación del software dentro del modelo de la arquitectura. En este contexto, los simuladores de instrucciones (*Instruction Set Simulators* (ISS)) constituyen el elemento central de una plataforma virtual. Existen diversas técnicas que se pueden usar para simular el comportamiento de un procesador. Como se ha comentado previamente, es preciso conseguir un balance adecuado entre la velocidad de la cosimulación y la precisión en el modelado del comportamiento interno del procesador. Concretamente para un desarrollo temprano del software de *boot* es suficiente con un modelo funcional sin entrar en detalles de temporización internos. De hecho, algunos elementos de la arquitectura interna como puede ser la memoria cache, son frecuentemente deshabilitados en sistemas de tiempo real. En este contexto, las diferentes implementaciones de los ISS pueden dividirse en dos grandes grupos:

- Interpretación binaria del código.
- Traducción binaria a código nativo.

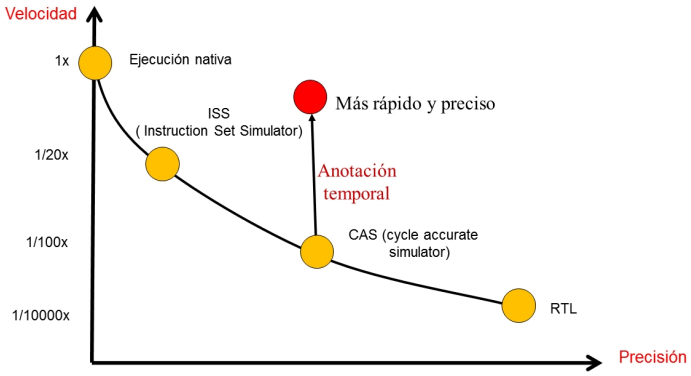


Figura 2.11: Precisión en el modelado de un procesador

La interpretación binaria del programa simula las fases de lectura, decodificación y ejecución de la instrucción. Permite modelar el comportamiento del procesador con la precisión que se necesite en cada fase del diseño software. Por contra, es la que introduce un tiempo de simulación mayor.

Suponiendo un entorno sin fallos, la velocidad de la cosimulación se puede mejorar usando técnicas de pre-decodificación de las instrucciones de forma que la decodificación previa del mismo tipo de intrucción se puede reutilizar varias veces durante la ejecución del código. Esta técnica se conoce genéricamente como *just-in-time cache compiled* (JIT-CC) [NBS+02, RMD03].

En la traducción nativa binaria se realiza una traducción del binario de la aplicación, ejecutado por el procesador simulado, al código de la máquina anfitrión con el consiguiente aumento de velocidad de ejecución [JT09]. La traducción puede realizarse de forma

estática previa a la ejecución o dinámica durante la ejecución. La complejidad de la traducción y la consiguiente mejora de las prestaciones está directamente relacionada con la semejanza entre los repertorios de instrucciones del procesador simulador y el anfitrión donde se ejecuta la traducción [You07]. Tienen en contra que la implementación no es portable y no proporciona información de la temporización de la instrucción original.

Aunque el esquema de interpretación binaria del código es el más lento en cuanto a cosimulación también es cierto que es el más realista cuando se pretenden modelar fallos en memoria o buses. Por ello se usan técnicas de anotación de parámetros no funcionales como pueden ser el tiempo de ejecución o el consumo energético de las operaciones internas del procesador, a la vez que se realiza un modelado funcional del repertorio de instrucciones, como se muestra en la figura 2.11. De esta forma se consigue una velocidad de cosimulación aceptable a la vez que se pueden obtener estimaciones de los parámetros no funcionales.

2.5. Inyección de fallos en modelos TLM2

La perturbación del comportamiento de un componente TLM2 puede realizarse internamente o en su interfaz TLM2 con el resto del sistema. El acceso al estado interno puede realizarse dotando al componente de un puerto de depuración y accediendo a él mediante la primitiva `transport_dbg`. En este caso, los fallos introducidos son dependientes del tipo de componente. Si la inyección se realiza en la interfaz entre dos componentes, desde el punto de vista de la verificación software, la información que hay que corromper es la relacionada con los buses de direcciones y/o datos. Los campos de la transacción relacionados con esta información son `Address` y `Data`.

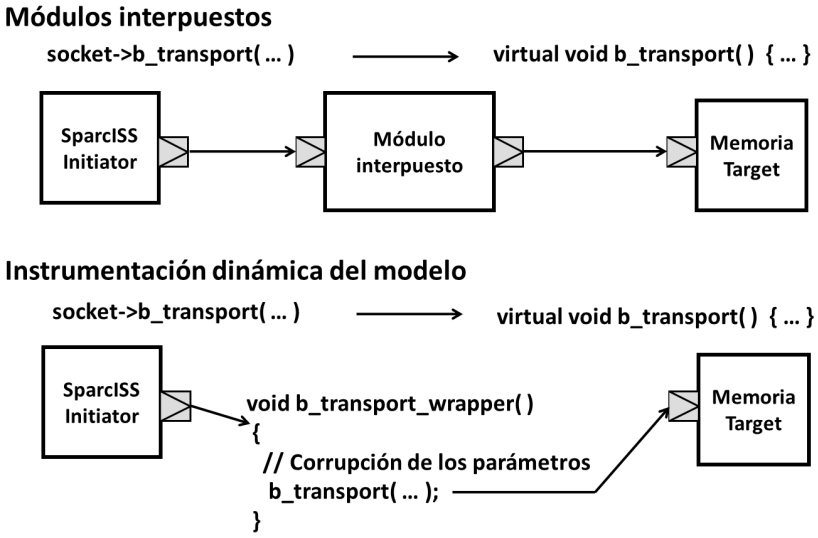


Figura 2.12: Instrumentación de la interfaz TLM2

2.5.1. Instrumentación dinámica de la interfaz TLM2

En SystemC, durante la fase de elaboración del sistema, se crean los diferentes módulos y sus puertos de conexión conocidos como *sockets*. Una vez creados los módulos, tiene lugar el procedimiento de *binding*, en el cual los *sockets* TLM2 de los iniciadores se conectan con los elementos equivalentes de los destinos de las transacciones.

La figura 2.12 muestra la interconexión de un módulo iniciador SparcISS con un módulo *target* que implementa una memoria. El esquema clásico para interceptar las transacciones entre los dos módulos es insertar un módulo de interposición. Este esquema duplica el número de puertos y conexiones y, además, el conexionado debe ser realizado en tiempo de compilación. La instrumentación dinámica de código tiene como objetivo añadir código que intercepte llamadas en tiempo de ejecución. El código interpuesto puede ser añadido o eliminado a voluntad durante la ejecución del sistema y su objetivo puede ser múltiple.

La instrumentación dinámica de código (*Dynamic Binary Instrumentation* (DBI)) es una técnica para interceptar llamadas a funciones, con el fin de analizar el comportamiento de un programa en tiempo de ejecución sin modificación del código fuente. Esta técnica se basa en la modificación binaria del comienzo del código de la función que se pretende interceptar con una instrucción de salto a otra función definida por el usuario que actúa como envoltorio (*wrapper*) de la función original. Esta rutina tendrá las funcionalidades que el usuario quiere agregar.

Para realizar la inserción de los *wrappers* existen distintas técnicas. Puesto que el lenguaje usado en SystemC/TLM es C++ se aprovecharán aspectos concretos de implementación binaria del código escrito en C++. Concretamente, tal como se muestra en la figura 2.13, se usará el esquema usado para la implementación de métodos virtuales.

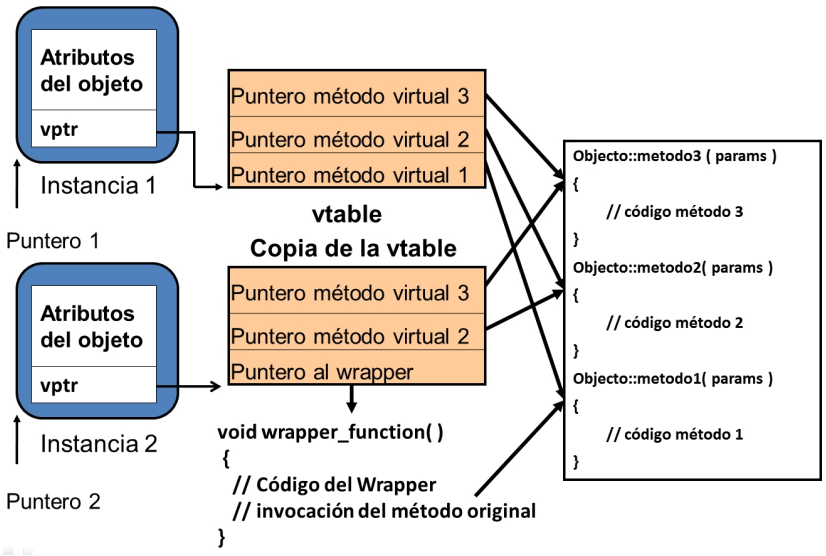


Figura 2.13: Métodos virtuales en C++

Cada vez que se crea una clase final que contiene métodos virtuales o que hereda de una clase base que contenga métodos virtuales, el compilador crea una tabla (VTABLE) con las direcciones de comienzo de las implementaciones concretas de los métodos virtuales para esa clase, véase la figura 2.13. Esto significa que cuando se invoca un método, la dirección de comienzo del mismo es obtenida en tiempo de ejecución. Este esquema es usado para la implementación del polimorfismo en C++. Es posible modificar la VTABLE para un objeto concreto de una clase e insertar la dirección de un código diferente al original que realice algún tipo de procesamiento con diferentes intenciones.

Aunque el método propuesto funciona solamente para métodos declarados virtuales, ésta es una situación muy común. En una jerarquía de objetos es muy común encontrar clases base cuyo único objetivo es describir una interfaz y servir como punto de arranque de la jerarquía para las clases derivadas. Todas las interfaces de componentes en SystemC están construidas siguiendo este esquema. En [dSS09a] se presenta un uso de la misma técnica para insertar "saboteadores" en el camino de las señales en una descripción SystemC, sin modificaciones en la descripción del código fuente del sistema, tal como se muestra en la figura 2.14.

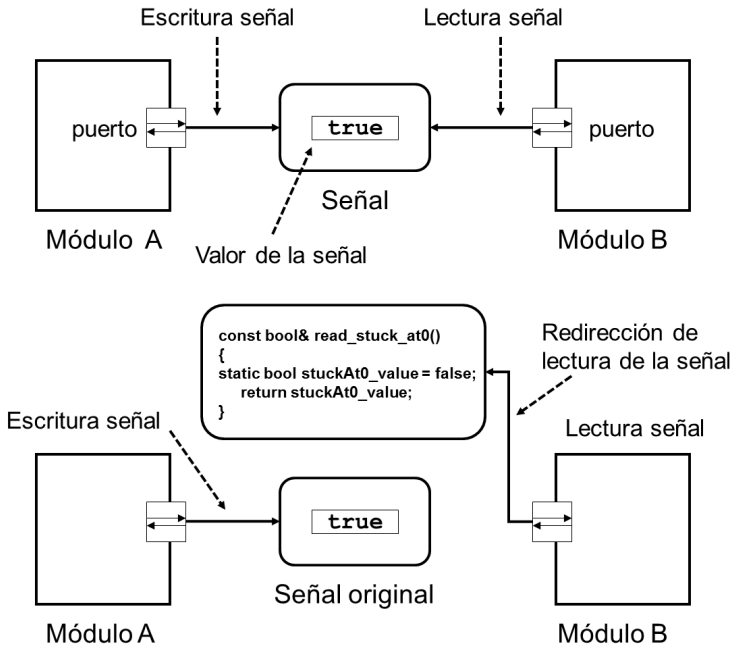


Figura 2.14: Inserción de un soboteador *stuck-at-0*

2.6. Plataformas virtuales

Las plataformas virtuales proporcionan a los desarrolladores una gran capacidad de control y visibilidad del diseño, aspectos imposibles por otros medios. Cada situación o condición puede ser provocada y convenientemente probada. Por ello, el uso de plataformas virtuales para el modelado de sistemas, exploración del espacio de diseño y desarrollo temprano del software, ha experimentado un incremento sustancial en los últimos años. El campo de investigación actual es el uso de modelos para la verificación y certificación del software mediante técnicas de inyección de fallos en plataformas virtuales [CN13]. Concretamente, el dominio de aplicación que ha apreciado un mayor auge en el uso de plataformas virtuales es el sector de la automoción debido al desarrollo de las *Engine Control Unitss* (ECUs) [BLQ14].

A diferencia de la ejecución simbólica de código, las técnicas experimentales permiten la verificación del software en tiempo de ejecución en un escenario practicamente equivalente al real. Usando SystemC/TLM2 es posible modelar sistemas HW/SW y observar su comportamiento en presencia de fallos. Por ejemplo, [SKKM13, CAMARC+11] usan esta metodología para el diseño y prueba de sistemas tolerantes a fallos implementados en FPGA. [ZKK11, GMC+12, LR13] usan la misma aproximación para la verificación de software de sistemas embebidos en red, tiempo antes de que el hardware este disponible.

Desde un punto de vista industrial, [MGA+12] describe la mejora de un entorno virtual previo que muestra el uso de plataformas virtuales en un desarrollo de un vehículo eléctrico híbrido. [HOC+12] presenta el codiseño y coverificación de un sistema donde se combina ejecución nativa de código con un simulador con precisión de ciclo. [RBM+14, RBM12, WWZ12, BvL14] presentan experiencias similares en entornos relacionados con la aviónica o espaciales. Relacionado con la verificación de la tolerancia a fallos de un sistema, [CC07] presenta un ejemplo de inyección en modelos atemporales basados en canales FIFO TLM1. [EHK13] propone mecanismos de endurecimiento para protocolos de comunicación entre componentes usando interfaces TLM2.

Además de las técnicas concretas de simulación en el nivel de transacción empleadas en difererentes dominios de aplicación, un campo de investigación activo es el desarrollo de metologías de codiseño y coverificación mediante el uso de plataformas virtuales [Gru13].

2.6.1. Plataformas virtuales comerciales

En el mercado existen una amplia variedad de empresas que comercializan herramientas para el modelado de sistemas y plataformas virtuales para la exploración del espacio de diseño y desarrollo temprano del software en sistemas embebidos. A continuación se presenta una lista de dichas compañías con los nombres comerciales de sus productos:

- Cadence [Sys14]: el entorno de desarrollo “Incisive” permite el modelado de sistemas embebidos usando SystemC/TLM2.

- Carbon [CDS14]: el entorno “SoC Designer framework” permite el diseño y depuración de software en etapas tempranas. Su núcleo de simulación está basado en SystemC y soporta interfaces TLM2.0.

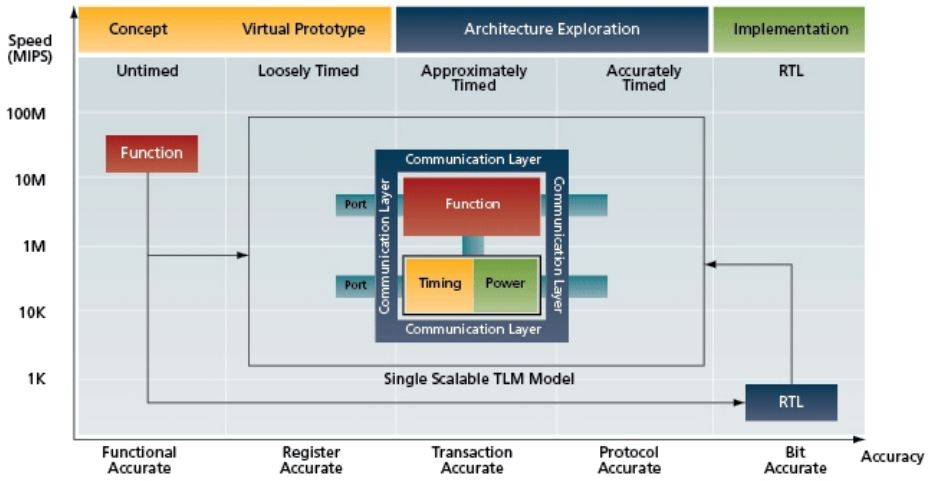


Figura 2.15: El flujo "ideal". Adaptación de figura tomada de Mentor Graphics

- CoWare [Inc14a]: la plataforma de desarrollo de CoWare puede dividirse en dos partes: "Platform Architect" para el modelado de las plataformas virtuales y "Platform Analyzer" para la ejecución y depuración del software en la plataforma.
- Imperas [Imp14]: Imperas proporciona un núcleo de simulación propietario basado en modelos de procesadores con traducción dinámica del código. Dispone de una interfaz capaz de incorporar módulos SystemC/TLM. En 2008 Imperas comenzó el programa "Open Virtual Platforms (OVP) initiative" [OVP14] e hizo públicos los modelos de simulación de los componentes pero mantiene propietario el núcleo de simulación.
- Mentor Graphics [Inc14b]: el entorno de desarrollo "Vista Architect" permite el modelado de plataformas virtuales mediante interfaces TLM2.
- ASTC [Com14]: el entorno de simulación y modelado de sistemas "ASTC/VLAB" Works está especialmente orientado al sector de la automoción.
- Synopsys [Inc14c]: el entorno integrado "Innovator" está basado en SystemC y puede usarse para el diseño de plataformas virtuales para desarrollo de sistemas embebidos.
- VirtuTech2 [Inc14d]: el entorno de simulación "Simics" está basado en un núcleo de simulación propietario. Ofrece una interfaz para integrar modelos externos con interfaz TLM2.0
- TSIM [Res13]: TSIM es un simulador propietario con precisión de ciclo de sistemas LEON2/3. Permite la ampliación de la periferia conectada al bus mediante una interfaz de programación propietaria. Ofrece la capacidad de inyección de fallos de acuerdo a patrones estadísticos.

La figura 2.15 muestra los aspectos "ideales" de la metodología TLM, en la cual, además de los aspectos funcionales es posible etiquetar el tiempo y el consumo, con la intención de realizar estimaciones del consumo de potencia en función de la arquitectura y del software. Todas las herramientas presentadas constan básicamente de dos partes. Una librería de componentes para realizar la composición de la plataforma virtual y un entorno gráfico de diseño y simulación del sistema. Para realizar la composición de una plataforma virtual es necesario disponer de los modelos de todos los componentes que la integran: procesadores, memorias y elementos de entrada/salida. Si un componente no está disponible es posible ampliar la biblioteca mediante la implementación del mismo y su integración con el conjunto usando interfaces TLM2.0. Por supuesto la compra del modelo al fabricante o a terceras partes también es posible.

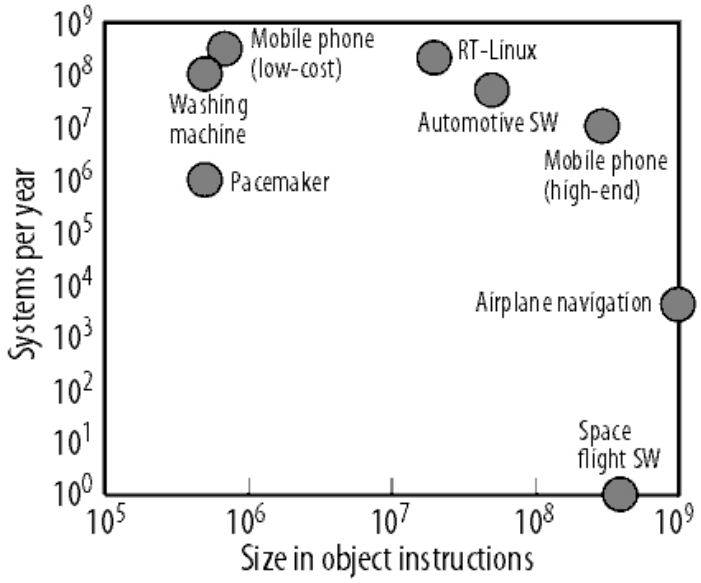


Figura 2.16: Número de unidades vs complejidad [EJ09]

Sin embargo el desarrollo de sistemas espaciales no tripulados tiene un problema añadido que se refleja en la figura 2.16. Estos sistemas tienen una complejidad software superior a los sistemas embebidos en automoción y con un orden de complejidad similar a la presente en un telefono *smartphone* actual. Los sistemas embebidos en aviónica comercial tienen una complejidad ligeramente superior. Pero a pesar de tener una complejidad de una magnitud semejante, un sistema espacial es siempre un prototipo del cual solo se ponen en circulación unas pocas unidades, la mayoría de las veces solamente una, de forma que el precio por unidad desarrollada es muy alto. Las herramientas comerciales presentadas previamente tienen un alto coste y sus licencias de uso suelen estar limitadas por tiempo de uso y/o por el número de estaciones de trabajo donde se puede instalar. Por otro lado, en términos de procesadores e interfaces de comunicaciones, las bibliotecas de componentes ofrecen mayoritariamente soporte para procesadores ARM e interfaces

de red *Controller Area Network* (CAN). En el caso de Solar Orbiter los requisitos son procesadores LEON2 e interfaces de red SpaceWire. En este contexto, usando una hipotética herramienta comercial sería necesaria la codificación ad-hoc de los modelos más complejos para su integración posterior en la interfaz gráfica. El uso de entornos abiertos como SimpleScalar [Sim14, AE02], sufre de los mismos problemas. No tiene soporte para procesadores SPARC y su objetivo principal está más orientado al modelado de la arquitecura interna del procesador que a la cosimulación del software embebido en el sistema final.

Capítulo 3

Desarrollo de la investigación

Maestro Yoda : «¡Tú siempre con tus "NO PUEDE HACERSE"! ¿Es que escuchándome no estabas?»

El Imperio Contraataca

3.1. Introducción

El Grupo de Investigación del Espacio de la Universidad de Alcalá está al cargo del diseño e implementación de la ICU del instrumento EPD embarcado en Solar Orbiter. Una descripción del proceso de codiseño HW/SW llevado a cabo en el diseño de la ICU puede encontrarse en el apéndice A.3 [SPP+13]. La unidad central de proceso está basada en un LEON2, procesador RISC de 32 bits sintetizable en FPGA y conforme a la arquitectura IEEE-1754 (SPARC V8). El núcleo del procesador está disponible en VHDL y es altamente configurable. Los requisitos del software de arranque de la ICU, de aquí en adelante BOOTSW, deben ser verificados en las etapas más tempranas del diseño, ya que afectan al software del sistema en su conjunto, al ser el responsable del arranque del sistema y del primer intercambio de mensajes de telemetría y telecomando (TM/TC) con la nave. Además, los requisitos de tolerancia a fallos exigen un proceso exhaustivo de verificación del proceso de arranque y la posible corrupción de los binarios de aplicación almacenados en las EEPROM. También es preciso verificar el correcto estado de las áreas SDRAM donde se despliegan los programas. Desde el punto de vista del desarrollo temprano de un software dependiente del hardware, como es el caso del BOOTSW, el primer problema es la ausencia de un hardware donde probar y depurar las primeras versiones del software. Otros aspectos están relacionados con el tipo de software que hay que desarrollar, concretamente un cargador (*bootloader*). En estos casos, parte del software ha de escribirse en el ensamblador propio del procesador y durante el arranque todavía no hay servicios de depuración que se puedan usar, por lo que la observabilidad es prácticamente nula. Para dar solución a este problema, se ha definido un modelo TLM2 del hardware básico de la ICU que ha servido para codificar una plataforma virtual denominada "Leon2ViP" con capacidad de inyección de fallos en memoria, donde se ha podido comenzar el desarro-

llo y depuración del BOOTSW independientemente de la disponibilidad del hardware y acortar los tiempos de integración cuando el hardware ya estuvo disponible.

La elección de los componentes se ha realizado de acuerdo con el análisis del ambiente espacial donde Solar Orbiter se va a desplegar [Tea10, Sor10, LD11, JCS+14]. Concretamente y relacionados con la posibilidad de SELs, la FPGA y los módulos de memoria tienen las siguientes especificaciones:

- Umbral LET de la FPGA para SEL: 117 MeV cm^2/mg.
- Umbral LET de los módulos EEPROM para SEL: 80 MeV cm^2/mg.
- Umbral LET de los módulos SDRAM para SEL: 80 MeV cm^2/mg.

3.1.1. Requisitos de tolerancia a fallos del BOOTSW

El principal requisito del BOOTSW es estar libre de errores funcionales puesto que al ser el único componente en PROM no puede ser actualizado una vez en órbita. Precisamente una de sus capacidades principales es proporcionar los mecanismos de regrabado de los binarios de aplicación en EEPROM. Esto exige una prueba exhaustiva del proceso de arranque, del enlace de comunicaciones SpaceWire con la nave y de los mecanismos básicos de actualización.

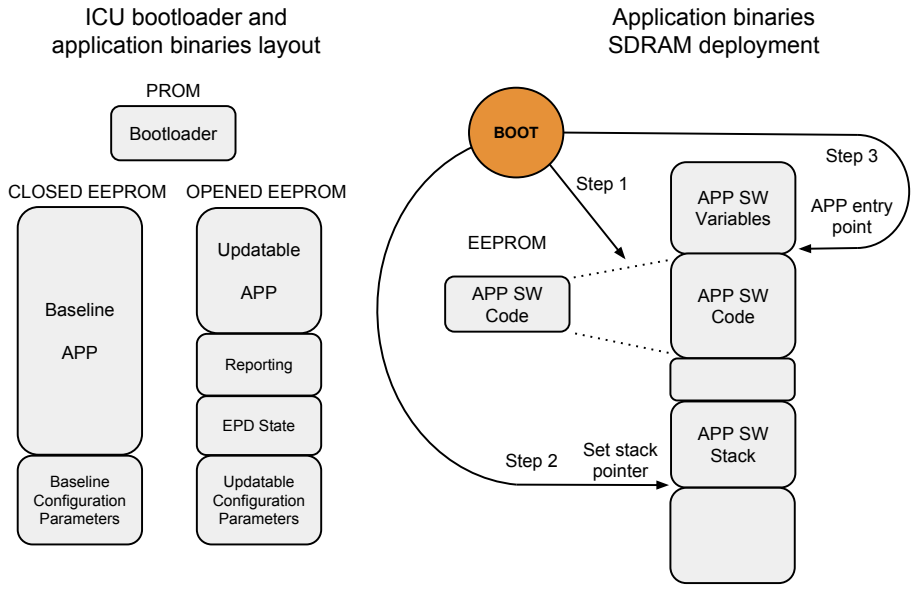


Figura 3.1: Mapa de memoria de la ICU

Como se muestra en la figura 3.1, el conjunto de memoria está organizado en tres bancos: PROM, EEPROM y SDRAM. La PROM almacena el BOOTSW y realiza las verificaciones de integridad de los demás bancos de memoria. El conjunto de las EEPROM

está organizado en dos bancos y contiene dos versiones del código de aplicación, conocidas como "baseline" y "updatable". La primera versión está en un banco cerrado donde la escritura está prohibida, mientras la segunda permite su actualización. Finalmente, la SDRAM almacena el binario durante la ejecución. La SDRAM está protegida contra SEUs mediante el uso de mecanismos EDAC y refresco de la memoria mediante *memory scrubing*.

Conocidos los valores umbrales para SELs señalados previamente, el riesgo principal consiste en la presencia de fallos *stuck-at* en áreas de los bancos de memoria EEPROM y SDRAM. El BOOTSW debe en primer lugar encontrar una región en SDRAM sin errores, suficiente para alojar la pila inicial del sistema. A continuación, verificar la integridad de los binarios almacenados en EEPROM y de las zonas de despliegue de los mismos en SDRAM. Si todo es correcto, cederá el control al software de aplicación. En caso de exitir algún problema se enviará por SpaceWire un mensaje de telemetría indicando la circunstancia y se quedará a la espera de indicaciones por parte del ordenador central de la nave. Entre las posibles acciones solicitadas puede estar el regrabado de la versión "updatable" con el fin de evitar las zonas dañadas. La verificación de la secuencia de arranque se encuentra detallada en el apéndice A.5 [dSSPP14].

3.2. Modelado de la ICU

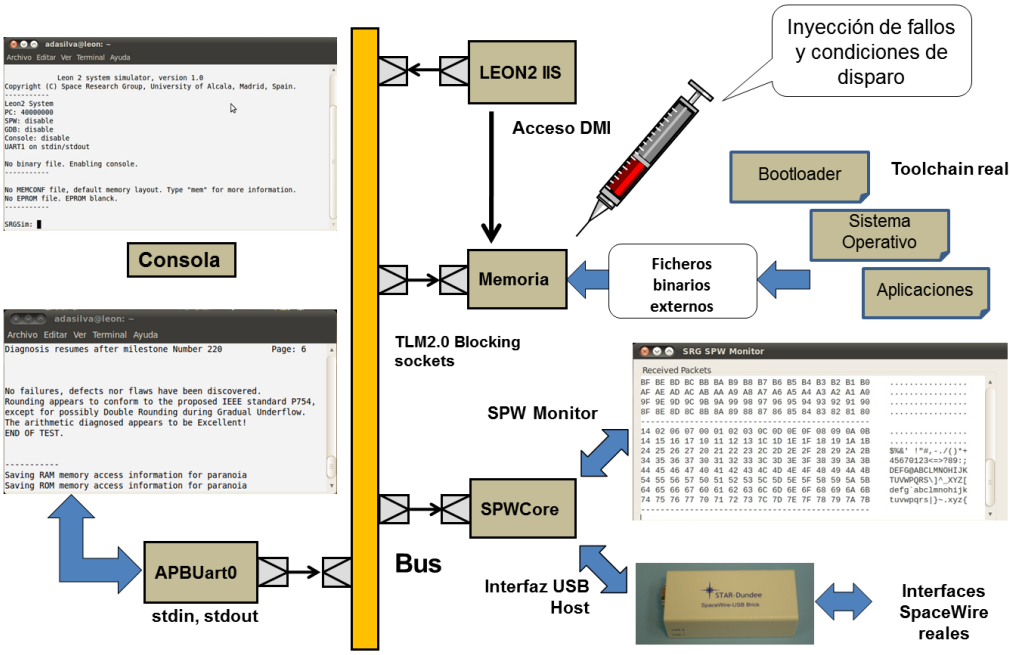


Figura 3.2: Arquitectura de "Leon2ViP"

Como trabajo de esta tesis se ha desarrollado un modelo TLM2 de un sistema basado en LEON2 con interfaces TLM2 bloqueantes y un estilo de programación temporal poco preciso, *loosely timed*, suficiente para el desarrollo del software de *boot*. El modelo incluye una interfaz SpaceWire real a través de un hardware específico. Además, el modelo permite la inyección de fallos en las interfaces TLM2, de forma que se puede verificar el comportamiento del software en presencia de errores en memoria. La figura 3.2 muestra los componentes principales de la plataforma virtual "Leon2ViP":

- LEON2 ISS: Simulador de Instrucciones SPARC V8. Ha sido desarrollado a partir de un modelo LEON3 previo descrito en el apéndice A.2 [dSS11b].
- Memoria: Emula los bloques de memoria PROM, EEPROM and SDRAM. El mapa de memoria es completamente configurable mediante un fichero de configuración externo.
- SPWCore: Interface SpaceWire para la comunicación con el ordenador central de la nave. Esta interfaz puede ser virtual o estar físicamente *mapeada* sobre un hardware basado en un Star-Dundee USB SpaceWire Brick. El desarrollo y prueba de este core está descrito en la sección 3.5.
- APBUart0: Interfaz serie. Es usada por la entrada/salida estándar y puede ser *mapeada* sobre una interfaz serie real.

Las áreas PROM y SDRAM típicamente se rellenan desde el exterior mediante el uso de una orden de carga. "Leon2ViP" también emula el comportamiento específico de las áreas EEPROM. Las memorias EEPROM implementan los mecanismos *Software Data Protection* (SDP), de forma que la escritura está prohibida, pero puede ser habilitada mediante la escritura de una secuencia de habilitación. Al finalizar la ejecución, el contenido es guardado en ficheros de manera que al volver a arrancar se recupere la última información almacenada.

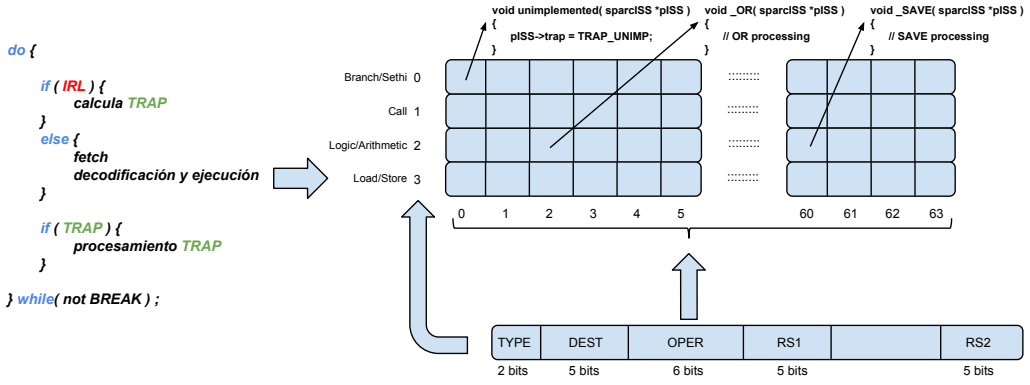


Figura 3.3: Bucle principal y decodificación en el ISS

La figura 3.3 describe el bucle principal de ejecución del ISS de "Leon2ViP". Para cada código de operación SPARC se accede a una tabla de punteros a funciones donde

se realiza la operación con los parámetros indicados en la instrucción. El hecho de que el repertorio de instrucciones siga una filosofía RISC ayuda a implementar este esquema ya que la intrucciones tienen una estrucutura y longitud fija. Para cada iteración del bucle se pregunta por la existencia de una interrupción externa, *Interrupt Request Level* (IRL). En este caso se calcula la entrada en la tabla de interrupción (TRAP) a la que se transferirá la ejecución. Esta circunstancia también podría ser provocada por la existencia de algún tipo de excepción en la ejecución normal de una instrucción. Aunque el bucle descrito previamente se ejecuta a la velocidad de la máquina anfitrion donde se esté ejecutando “Leon2ViP”, pueden existir puntos de sincronización de tiempo real programados en los temporizadores de LEON2. En la implementación que se ha realizado de los temporizadores de LEON2, se traslada la configuración que realiza el software embebido a temporizadores reales del sistema operativo anfitrion. De esta forma si desde un sistema operativo invitado se programa un *tick* cíclico, por ejemplo para establecer una rodaja de tiempo, se provocará una interrupción en tiempo real exactamente con dicha temporización. Ello permite ejecutar sistemas operativos multitarea como RTEMS o eCos como se muestra en el sección 3.4.3.

3.3. Interfaz de usuario de “Leon2ViP”

3.3.1. Opciones de línea de comandos

Se pueden especificar diversos “switch” en la línea de comandos:

- **-leon3** : Se construye un sistema LEON3. Por defecto se construye un sistema LEON2.
- **-file** <binario SREC>: Se carga el binario indicado y se ejecuta sin mostrar la consola de comandos.
- **-pc** <valor hexadecimal> : Valor inicial del registro PC, NPC se inicializa a PC+4. Por defecto PC se inicializa a 0x40000000.
- **-batch** <fichero de comandos>: Especifica un fichero de comandos que se ejecutará en lugar de mostrar la consola de comandos.
- **-spw** <brick/monitor>: Habilita el controlador de SpaceWire en el mapa de memoria. Es preciso indicar si la salida se envían al monitor software o el USB SpaceWire Brick.

Si no se especifica ningún “switch” se arranca un sistema LEON2

```
./leon_vip
```

y se muestra una ventana semejante a la mostrada en la figura 3.4.

3.3.2. Mapa de memoria

El mapa de memoria es configurable mediante un fichero externo con un formato como el mostrado a continuación:

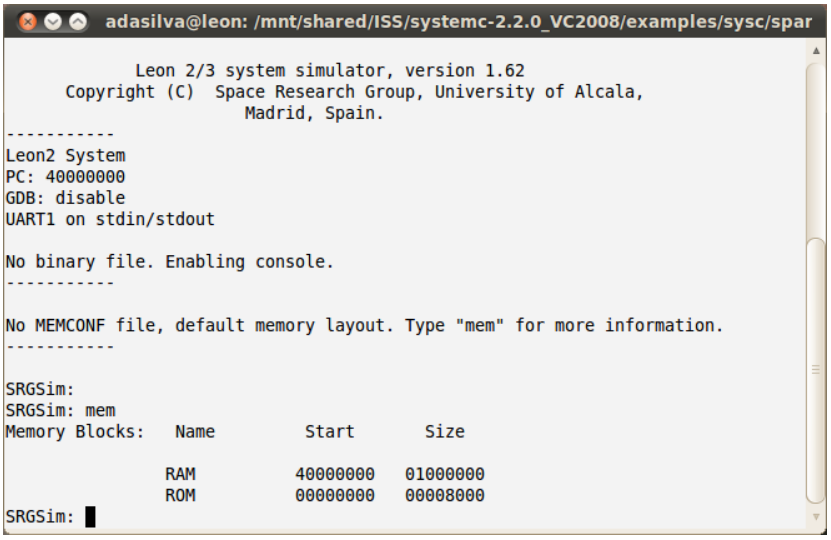


Figura 3.4: Consola de "Leon2ViP"

```
# Memory layout definition file
#
# Each line define a memory block, expected fields are:
# NAME: Memory Block Name, used by "mem" command
# START ADDRESS: Memory Block start address
# SIZE: Memory Block size in bytes
# ATTRIBUTES: R (readable), W (writable), RW (both),
#             S(static)=contents are saved/loaded from
#                       a file named "name.bin"
# Example default configuration:
# RAM   0x40000000 0x01000000 RW
# EPROM 0x20000000 0x00100000 RWS
# ROM   0x00000000 0x00008000 R
#

RAM    0x40000000 0x10000000 RW
#EPROM  0x20000000 0x00100000 RWS
EPROM0 0x20000000 0x00080000 RWS
EPROM1 0x30000000 0x00080000 RWS
ROM    0x00000000 0x00008000 R
```

En caso de no existir el fichero de configuración se genera un mapa por defecto. En cualquier caso con el comando "mem" se puede ver el mapa que se tiene definido. En la figura 3.4 se muestra el mapa por defecto en caso de no existir el fichero de configuración.

3.3.3. Operaciones básicas

El simulador muestra una línea de petición donde el usuario podrá teclear diferentes comandos que se describen brevemente a continuación.

3.3.3.1. Carga de programas

- **load** <binario SREC> : Realiza la carga en memoria de un fichero binario en formato SREC.
- **quit** : Sale del programa.

3.3.3.2. Control de ejecución

- **run** <tiempo> : Carga PC con 0x40000000 y NPC con 0x40000004. Lanza la ejecución del programa durante el tiempo establecido. Si no se especifica tiempo alguno se entiende que la ejecución es por tiempo indefinido. La ejecución podrá detenerse antes en el caso de alcanzar algún punto de ruptura.
- **continue** <tiempo> : Continúa la ejecución con el PC y NPC actual. El parámetro “tiempo” funciona igual que en el caso anterior.
- **step** : Ejecuta la instrucción actual indicada por PC.
- **break** <dirección> : Situa un *breakpoint*, punto de ruptura en acceso a código, en la dirección indicada. Al llegar PC a este punto se detiene la ejecución. Se avanza ejecutando el comando “step”.
- **watch** <dirección> : Situa un *watchpoint*, punto de ruptura en acceso a datos, en la dirección indicada. Al acceder a esta dirección se detiene la ejecución. Se avanza ejecutando el comando “step”.

3.3.3.3. Visualización/modificación del estado

- **mem** :
  - sin parámetros muestra el mapa de memoria.
  - **mem** <dirección> muestra el contenido de la memoria a partir de la dirección indicada.
- **wmem** <dirección> <dato>: Escribe el dato en la dirección indicada.
- **reg** : visualiza/modifica el contenido de la ventana de registros actual.
  - sin parámetros muestra el contenido de los registros.
  - **reg g4** 0x12345678 guarda el dato en el registro indicado.
- **dis** <dirección>: desensambla el contenido de memoria a partir de la dirección indicada. Si no se especifica dirección usa el valor actual del PC.

3.3.3.4. Inyección de fallos

- **stuckat0** <dirección> <máscara> : Aplica la máscara sobre el contenido de memoria usando una operación AND. Simula el efecto de un bit erróneo que está a cero mientras el fallo esté activo. Permite el modelado de fallos intermitentes y permanentes.
- **stuckat1** <dirección> <máscara> : Aplica la máscara sobre el contenido de memoria usando una operación OR. Simula el efecto de un bit erróneo que está a uno mientras el fallo esté activo. Permite el modelado de fallos intermitentes y permanentes.
- **stuckat-last** <dirección> <máscara> : En las operaciones de escritura, mantiene el último valor almacenado en los bits indicados por la mascara. Permite el modelado de fallos intermitentes y permanentes.
- **stuckend** <dirección> : Elimina el error aplicado a la dirección de memoria.
- **bitflip** <dirección> <máscara> : Invierte el valor los bits de memoria indicados por la máscara. Permite el modelado de fallos transitorios. En su forma básica este comando solo inyecta un fallo pero acepta extensiones para la inyección de fallos de acuerdo a distribuciones estadísticas, véase la sección 3.6.2.

3.3.4. Funcionamiento en modo *Batch*

El modo de funcionamiento por defecto del simulador es el modo "consola". En este modo el usuario introduce los comandos que estime oportunos. Con el objeto de facilitar la integración del simulador en un entorno automático de test se ofrece el modo de funcionamiento "batch". En este modo de funcionamiento, todos los comandos descritos previamente pueden ser escritos en un fichero de comandos de forma que no sea necesario escribirlos de forma manual en cada sesión sino que pueden ser generados por una herramienta de configuración de acuerdo a un perfíl de prueba definido. Por ejemplo, supongamos un fichero `comandos.txt` con el siguiente contenido:

```
# no_user         # Si está activo no retorna control
                  # a la consola al alcanzar un breakpoint

load ./bootsw.srec
reg pc  0x00000000
reg npc 0x00000004
continue
```

Si se arranca el simulador:

```
./leon_vip_linux -batch comandos.txt
```

Se procede a la carga del binario, se inicializan los registros PC y NPC y se continúa la ejecución del código. En el caso de que haya algún punto de ruptura, el comportamiento por defecto de simulador es detenerse y devolverá control a la consola para que el usuario pueda inspeccionar el estado del sistema y continuar con la ejecución. También existe la

posibilidad de usar los puntos de ruptura como condiciones de disparo de fallos intermitentes o permanentes. En este caso hay que indicar que no hay que retornar control al usuario al alcanzarse el punto de ruptura, sino que se continúa con el procesamiento del fichero de comandos, véase la sección 3.6.

3.3.4.1. Ejemplo de punto de ruptura para depuración

Sea el siguiente programa básico de ejemplo donde se invoca una función desde el programa principal:

```
\#include <stdio.h>
                                400011a0 <diHola>:
                                400011a0: 9d e3 bf a0  save  %sp, -96, %sp
void diHola() {                 400011a4: 03 10 00 15  sethi  %hi(0x40005400), %g1
printf(‘‘HOLA...\n’’);          400011a8: 90 10 62 20  or  %g1, 0x220, %o0
}                               400011ac: 40 00 00 2e  call  40001264 <puts>
                                400011b0: 01 00 00 00  nop
int main() {                    400011b4: 81 e8 00 00  restore
                                400011b8: 81 c3 e0 08  retl
diHola();                       400011bc: 01 00 00 00  nop
return 0;
}                               400011c0 <main>:
                                400011c0: 9d e3 bf a0  save  %sp, -96, %sp
                                400011c4: 7f ff ff f7  call  400011a0 diHola
                                400011c8: 01 00 00 00  nop
                                400011cc: 82 10 20 00  clr  %g1
                                400011d0: b0 10 00 01  mov  %g1, %i0
                                400011d4: 81 e8 00 00  restore
                                400011d8: 81 c3 e0 08  retl
                                400011dc: 01 00 00 00  nop
```

En la columna de la izquierda se encuentra el código fuente y en la columna de la derecha un recorte del binario donde se aprecian las direcciones de carga del código y su desensamblado. Se ejecutará el programa usando el siguiente fichero de comandos:

```
load ../demos/bin/hola.srec
break 0x400011A0
run
```

Al ejecutar el simulador:

```
./leon_vip_win32 -leon3 -batch comandos.txt
```

El comportamiento se muestra en la figura 3.5. En ella se observa cómo se carga el ejecutable. El simulador busca un fichero de símbolos con el mismo nombre que el ejecutble pero con extensión *sym* que en este caso no se proporciona. A continuación se establece el punto de ruptura en la dirección 0x400011a0, comienzo de la función tal como se puede ver en el listado, y se lanza la ejecución. Al alcanzar el punto de ruptura se devuelve control al usuario, pudiendo observar el contenido de memoria y registros, ejecutar paso a paso o reanudar la ejecución. En este caso se muestra la ejecución del comando “dis”, el cual muestra el código a partir de la posición actual del contador de programa PC.

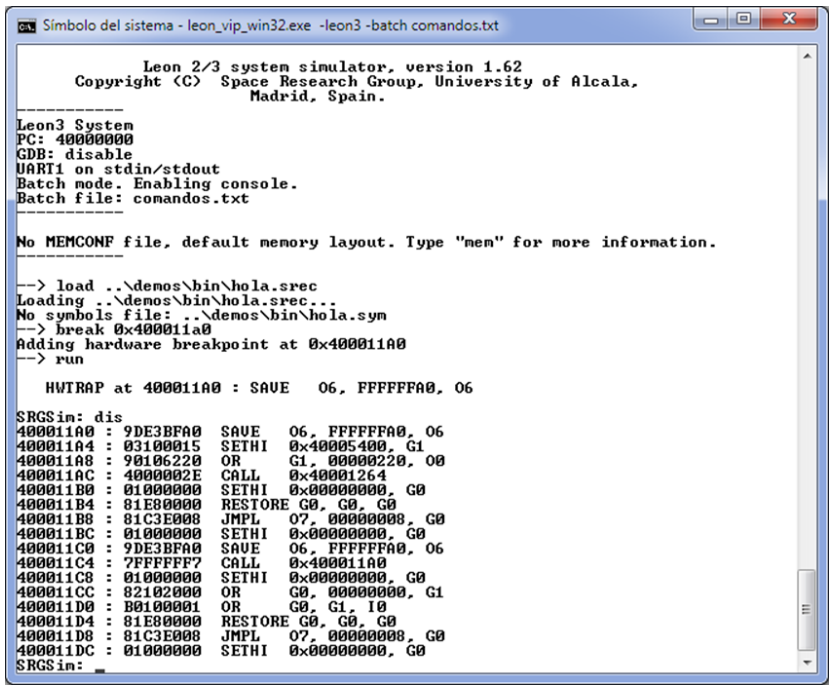


Figura 3.5: Ejemplo con *breakpoint* para depurado

3.4. Tests de la plataforma virtual

Para la verificación de la plataforma virtual se han realizado diversos test que han consistido en la ejecución de programas con diferentes características. Los primeros son tests sintéticos, esto es programas en ensamblador sin funcionalidad aparente, donde se ejercitan por separado cada una de las instrucciones máquina del procesador. En segundo lugar se han ejecutado *bechmarks* clásicos como Dhrystone o Stanford. Aunque existen test más modernos, estos se distribuyen con el compilador *sparc-elf-gcc* proporcionado por Gaisler Research [Res13] y que es usado en el proyecto para la compilación del código. Todos los tests se llevaron a cabo en un ordenador portátil genérico a 1,66 Ghz con 1 Gbyte de RAM bajo el sistema operativo Ubuntu 12.10. Los programas probados fueron:

- Paranoia Test: Paranoia caracteriza la implementación de las operaciones en coma flotante. Mediante el *switch* de compilación *-msoft-float* se usa la implementación software del estandar IEEE754 en vez del hardware específico. Este test hace un uso intensivo de las operaciones aritmético/lógicas y de comparación.

- Dhrystone Benchmark: Dhrystone pretende ser representativo de las prestaciones en procesamiento de cantidades enteras.

- Stanford Benchmark: Stanford contiene diez aplicaciones típicas y mide el tiempo de ejecución de cada una de ellas. Las aplicaciones pretenden ser representativas de diversos algoritmos de procesamiento usados en sistemas de telecomunicación.
- Hash *Secure Hash Algorithm* (SHA): El conjunto de funciones SHA son un conjunto de funciones criptográficas para el cálculo de valores resumen (*HASH*) desarrolladas por *National Security Agency* (NSA) y publicadas por *National Institute of Standards and Technology* (NIST). Los algoritmos SHA se usan comúnmente como verificadores de integridad de archivos en sustitución de *MD5*. En este test se usa una implementación abierta de SHA-256 en código C para generar los *HASH* de un conjunto de datos conocidos y se compara con los *HASH* de la misma información generados en una plataforma PC con otro software generador de resúmenes.
- Aplicaciones multitarea sobre sistemas operativos embebidos como eCos y RTEMS: Los tests anteriores verifican la correcta implementación del conjunto de instrucciones, así como el correcto acceso a memoria. Mediante el uso de sistemas operativos multitarea con planificación mediante ranura de tiempo se verifica el correcto funcionamiento del reloj de tiempo real y el procesamiento de interrupciones hardware.

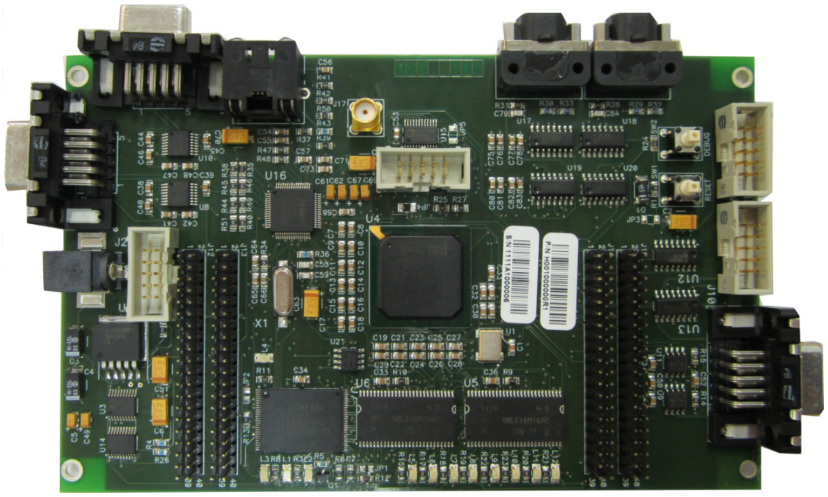


Figura 3.6: Tarjeta FPGA A3P

Las prestaciones proporcionados por "Leon2ViP" se han comparado con un sistema LEON2 real implantado en FPGA en una placa A3P [SRG12] a 20Mhz, lo que proporciona una capacidad de 17 MIPS. Esta frecuencia de reloj se ha escogido por ser la misma que se usará en la placa de procesamiento de la ICU. La placa A3P ha sido desarrollada por el grupo SRG, véase la figura 3.6, e incorpora el siguiente hardware:

- Actel ProAsci3E A3PE300-FG484 FGPA.
- 64 Mbytes de SDRAM, 8 Mbytes de FLASH.

- Interfaces eléctricas de comunicaciones
  - 2 RS-232 UARTs.
  - 4 RS-424 UARTs.
  - Fast Ethernet.
  - Dual CAN Bus.
  - 2 LVDS SpaceWire.
  - 4 x 40 I/Os de propósito general.

3.4.1. Resultados de la ejecución de Paranoia

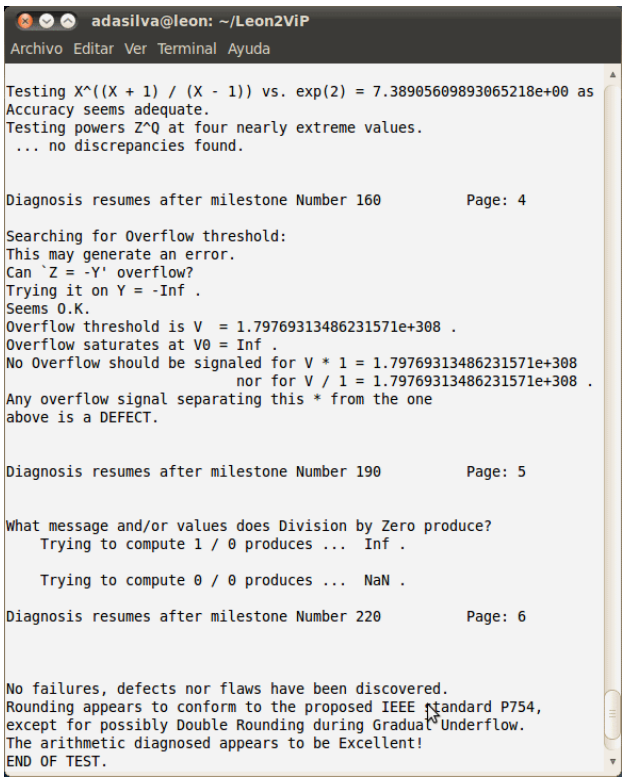


Figura 3.7: Ejecución del test Paranoia

Paranoia prueba la implementación de las operaciones en coma flotante por software y hace un uso intensivo de las instrucciones aritméticas y lógicas. De acuerdo a los resultados de la ejecución mostrados en la figura 3.7, la plataforma virtual se comporta de forma adecuada.

3.4.2. Resultados de los test Dhrystone, Stanford y SHA

Como se esperaba, "Leon2ViP" en una plataforma PC actual es más rápido que el sistema LEON2 a 20 Mhz en la placa A3P. Dependiendo de la prueba concreta, "Leon2ViP" es entre 1,5 a 5 veces más rápido que el sistema real ejecutándose en un ordenador portátil estándar. Ello muestra que hay capacidad suficiente para aumentar la precisión temporal de la simulación manteniendo la capacidad de cosimulación HW/SW. Pero desde el punto de vista de la verificación funcional de la plataforma virtual, el resultado de los tests no está tanto en la velocidad de ejecución como en la precisión de los resultados obtenidos. En el caso de *Dhrystone* se prueba el procesamiento de enteros usando diversas construcciones típicas de los lenguajes de programación como invocación de funciones, bucles, condiciones, y operaciones con *arrays*. No se han encontrado discrepancias funcionales. Los resultados se han resumido en la tabla 3.1.

Tabla 3.1: Resultados del test *Dhrystone*

	Leon2ViP	A3P
μS por *Dhrystone*	2,4	3,6
Dhrystones por segundo	423728,8	280898,9
Dhrystones MIPS	241,2	159,9

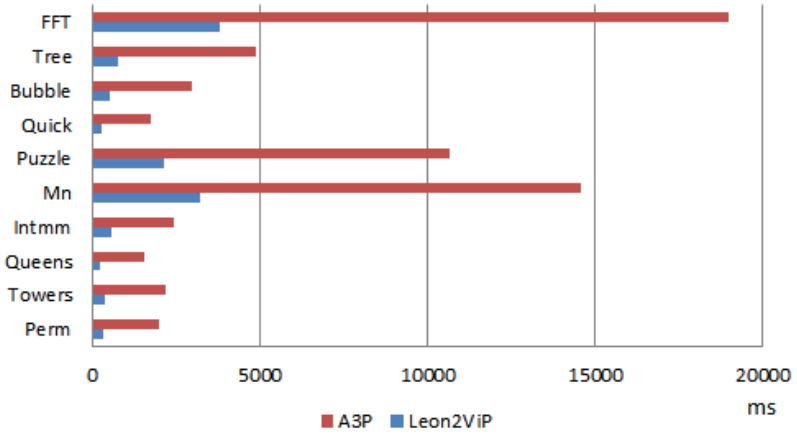


Figura 3.8: Comparativa temporal del test Stanford

Por otro lado Stanford realiza la ejecución de una serie de programas clásicos entre los que se encuentran:

- Towers: Resuelve el problema de las torres de Hanoi.
- Queens: Resuelve 50 veces el problema de ajedrez de las 8 reinas.
- Quicksort: Realiza la ordenación de un *array* usando el algoritmo `quick sort`.

- FFT: Realiza la transformada rápida de Fourier de una serie de datos.

En todos los casos se conoce el resultado esperado de la ejecución por lo que es posible establecer la correcta ejecución del código en la plataforma virtual. Puesto que no se ha detectado discrepancia alguna, estos resultados proporcionan una gran confianza en la implementación de "Leon2ViP". La comparativa temporal con el test Stanford se muestra en la figura 3.8.

3.4.3. Resultados de la ejecución de eCos/RTEMS

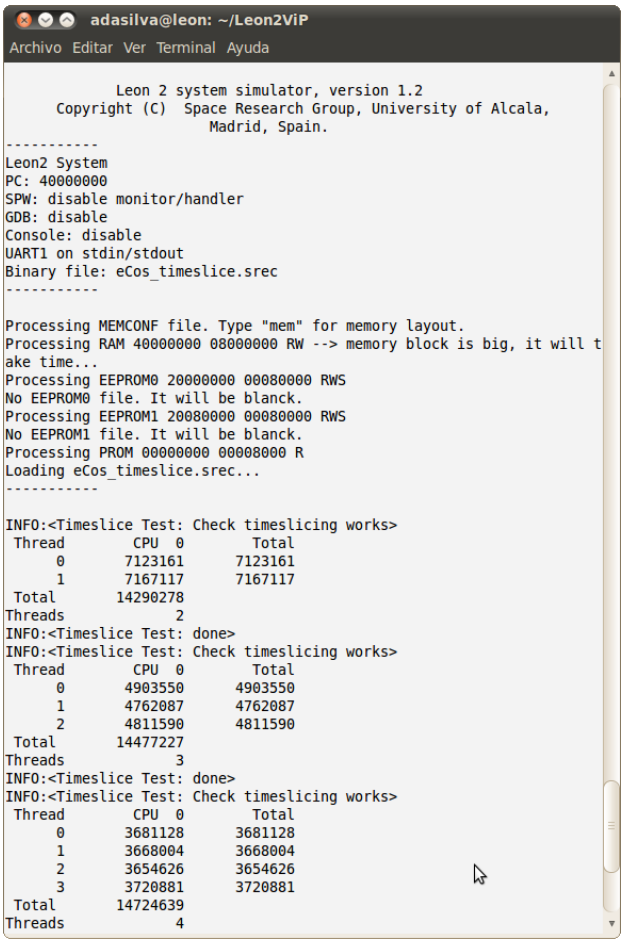


Figura 3.9: Ejecución de eCos en "Leon2ViP"

Como se detalla en la figura 3.9, se han ejecutado diversas aplicaciones multitarea sobre sobre sistemas operativos de libre distribución como eCos [eCo14] y RTEMS [RTE14]. En el caso de eCos se muestra la ejecución de un test de verificación del mecanismo

de planificación de tareas mediante ranura de tiempo. Este código se distribuye con el conjunto del sistema operativo. En el caso de RTEMS, se realiza la ejecución de un test con tres tareas con planificación *Rate Monotonic Scheduling* (RMS). No se han encontrado discrepancias y todos los plazos temporales especificados en los programas de test se cumplen.

3.5. Desarrollo y prueba de la interfaz SpaceWire

La comunicación de la ICU con el ordenador central del satélite se realiza mediante el envío de telemetrías y la recepción de telecomandos (TM/TC) a través de la interfaz SpaceWire. Es por tanto un elemento crítico en el desarrollo de la ICU. El desarrollo del conjunto HW/SW del core SpaceWire es un ejemplo práctico de codesarrollo mediante el uso de plataformas virtuales. Partiendo de la especificación inicial del comportamiento y de la interfaz de registros que ofrece al software, el desarrollo HW del core junto con su versión virtual dentro de la plataforma, corren en paralelo realimentándose mutuamente. El desarrollo del *driver* de dispositivo, software dependiente del hardware, puede realizar sus primeras pruebas de integración usando la plataforma virtual, aunque en un principio ésta solamente muestre un conjunto de registros *dummy* sin ningún tipo de comportamiento detrás.

El desarrollo del core SpaceWire se ha realizado a partir de un core más simple desarrollado en trabajos anteriores [CAP+10]. La versión desarrollada para la ICU incorpora un controlador tipo DMA y actúa como maestro de bus para el acceso directo a memoria. De esta forma, el software puede programar el envío o recepción de paquetes mediante el uso de descriptores de memoria y continuar con otras tareas mientra el controlador accede a la información de los paquetes. Desde el punto de vista de "Leon2ViP", implica que el core virtual tiene dos sockets TLM2, uno actúa como maestro de bus (*Initiator*) y el otro como esclavo de bus (*Target*), como se muestra en la figura 3.10.

Para la correcta visualización y depuración del proceso de envío y recepción de paquetes se dispone de un monitor gráfico que visualiza el contenido de los paquetes intercambiados por el BOOTSW a través de la interfaz SpaceWire. Aparte del envío de paquetes de TC de tipo DUMP y CHECK, el monitor permite la edición binaria de los paquetes, de forma que se pueda comprobar la correcta implementación de la decodificación de los paquetes recibidos por parte del BOOTSW. Además del monitor, usando la interfaz SpaceWire basada en un Star-Dundee USB SpaceWire Brick, es posible la comunicación física real con el *Electrical Ground Support Equipment* (EGSE). El EGSE es una herramienta esencial para la verificación del proceso de integración de todos los sensores desarrollados por otros equipos de trabajo con la ICU y su software de aplicación. Esta herramienta simula el comportamiento del ordenador central y se comunica con la ICU mediante la interfaz SpaceWire. Además de las operaciones básicas de control de la ICU se puede verificar el correcto funcionamiento de las operaciones relacionadas con los objetivos científicos como es la recolección, procesamiento y envío de los datos obtenidos de los sensores. De esta forma es posible probar el software real en un entorno VHIL.

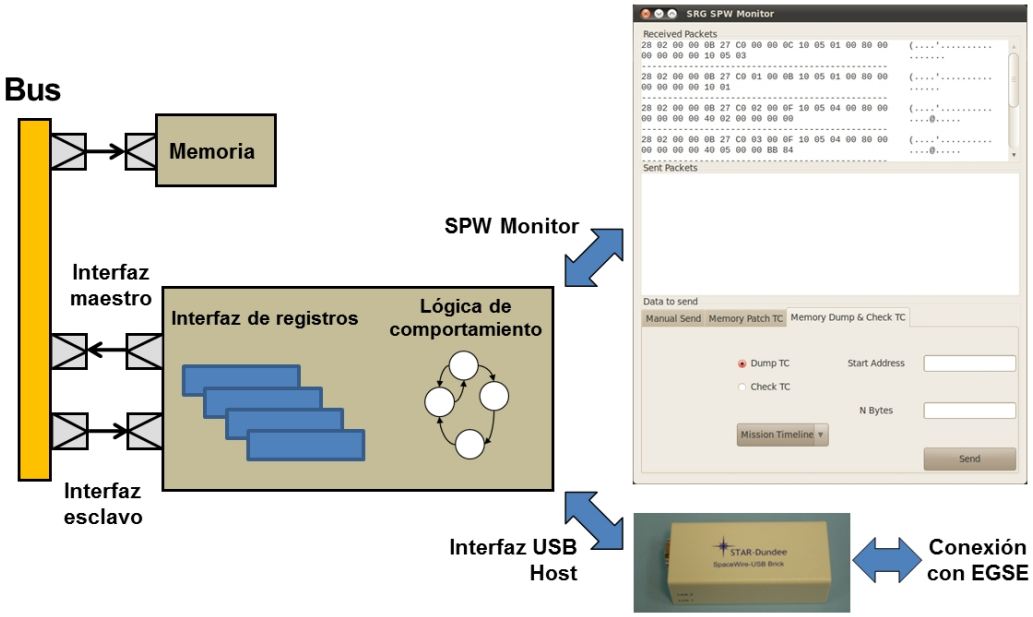


Figura 3.10: Core SpaceWire Virtual

3.6. Inyección de fallos en memoria

Además de servir como herramienta de desarrollo y depuración de los aspectos funcionales del software de *boot* de la ICU, "Leon2ViP" también ofrece la capacidad de inyectar fallos en memoria con la intención de simular los efectos del ambiente espacial. Para ello la consola de usuario ofrece una serie de órdenes que permiten insertar fallos transitorios, intermitentes y permanentes en localizaciones concretas de memoria.

3.6.1. Inyección de fallos manual y en modo "batch"

Los comandos descritos en el apartado 3.3.3.4 proporcionan los mecanismos básicos de corrupción de los contenidos de la memoria. El instante de inserción puede ser seleccionado mediante puntos de ruptura del código (*breakpoints*) o puntos de acceso a datos (*watchpoints*). La deshabilitación de los fallos *stuckat* puede hacerse también mediante los mismos mecanismos. De esta forma se pueden modelar fallos temporales o permanentes. Por ejemplo:

```
no_user                          # No se retorna control a la consola
                                 # al alcanzarse los puntos de ruptura

load ./bootsw.srec
reg pc   0x00000000
reg npc  0x00000004
```

```
stuckat0 0x20000000 0x7FFFFFFF # Stuckat0 del bit más significativo
                               # del primer octeto del primer banco de EEPROM
break    0x00014758            # <IMAGE_MANAGER_CHECK_PATCH_IN_RAM_LIMITS>
continue                       # Ejecuta el código hasta que se alcance un
                               # punto de ruptura o el fin del cóodigo.
stuckend 0x20000000            # Fin de la inyección
continue
```

En ejemplo anterior se activa un fallo "stuckat0" desde el comienzo de la simulación y se establece un punto de ruptura en una rutina del BOOTSW que en este caso actuará como *trigger* de desactivcación del fallo. Con la orden "continue" se arranca la ejecución con los valores actuales de PC y NPC. Alcanzado el punto de ruptura, se ejecutará el comando "stuckend" que desactiva el fallo definido previamente y se continúa con la ejecución.

3.6.2. Patrones estadísticos de inserción de fallos transitorios

La orden *bitflip*, tal como se describe en la sección 3.3.3.4 modela fallos transitorios que en el caso del modelo hardware de la ICU se solucionan medianta el uso de EDAC. El modelo TLM2 del bus ante la lectura de un bit erróneo soluciona el problema y opcionalmente informa al software mediante una excepción. En el caso de dos fallos consecutivos en la misma palabra, informa de un fallo no corregible. Este esquema permite la prueba de algoritmos de *scrubbing* [GTB+09, HSS12] al permitir inyectar fallos sobre la misma posición con las condiciones de disparo que se estimen oportunas.

Además de los esquemas de inyección descritos previamente, este comando puede usarse para definir patrones de fallos sobre un rango de posiciones de memoria, semejantes a los que se podrían producir en situaciones reales [JG09]. El formato es el siguiente:

- **bitflip** <dirección> range:<rango de direcciones> <average/uniform> <tasa> : Inyecta fallos transitorios en el rango indicado, la palabra de memoria se selecciona de forma aleatoria.
  - **average tasa** : Inyecta una "tasa" fallos de promedio. Los instantes concretos se calculan de acuerdo a un proceso *Poisson* homogéneo. Esto es la tasa no cambia con el tiempo.
  - **uniform tasa** : Inyecta una "tasa" fallos de uniforme por segundo. Los instantes concretos están uniformemente distribuidos en el intervalo de tiempo.
- **bitflip** <dirección> end : Detiene la inyección de fallos de acuerdo a un patrón establecida previamente.

Para la activación/desactivación de estos fallos se pueden seguir los mecanismos descritos previamente para los comandos "stuckat". Sucesivas invocaciones del comando "bitflip" con tasas de inyección diferentes permite el modelado de proceso *Poisson* no homogéneos. Por ejemplo, el siguiente fichero de comandos establece que al comienzo de la ejecución de una rutina en concreto se produce una incremento de la tasa de inyección, que se mantiene durante un segundo. Después de este tiempo se retorna al promedio de inyección original.

```
no_user                        # No se retorna control a la consola
                               # al alcanzarse los puntos de ruptura
```

```
load ./bootsw.srec
reg pc   0x00000000
reg npc  0x00000004
                              # Inyecta un promedio de 10 fallos
                              # en el rango 0x40000000-0x40000FFF
bitflip  0x40000000 range:0x1000 average 10

break    0x00014758           # <IMAGE_MANAGER_CHECK_PATCH_IN_RAM_LIMITS>

continue                      # Lanza la ejecución, no retorna hasta que
                              # se alcanza el breakpoint

                              # Inyecta un promedio de 50 fallos
                              # en el rango 0x40000000-0x40000FFF
bitflip  0x40000000 range:0x1000 average 50

continue 1s                   # Ejecuta el código durnate el tiempo indicado
                              # Retorna al cumplirse el periodo indicado

                              # Inyecta un promedio de 10 fallos
                              # en el rango 0x40000000-0x40000FFF
bitflip  0x40000000 range:0x1000 average 10

continue                      # Ejecuta hasta el final
```

3.6.3. Control de las condiciones de disparo mediante máquinas de estado

Uno de los aspectos más complicados a la hora de definir un campaña de inyección de fallos es poder definir un conjunto de fallos representativo que ejercite de forma efectiva aquellos elementos del sistema que se desean probar. En este sentido, la inyección de fallos basada en patrones estádisticos, aunque refleja la naturaleza real del sistema durante su explotación, es poco controlable ya que no permite elegir el instante y lugar de la inyección. De una forma semejante al problema de explosión de estados que ocurre en la verificación formal de modelos, el espacio de fallos se vuelve impracticable y es necesario reducirlo para lograr una inyección de fallos efectiva.

Una buena caracterización del modelo de fallos debe ser lo más versátil posible de forma que permita un amplio conjunto de combinaciones entre puntos de fallo, condiciones de disparo, tipo de fallo introducido, duración del fallo y patrones de repetición para el modelado de fallos intermitentes. Sería deseable que este comportamiento pudiera definirse en un lenguaje común e independiente de la plataforma de inyección. Por ejemplo en [dSMGC+10] se propone un esquema XML para la definición de conjuntos de fallos independiente de la herramienta de inyección.

Uno de los mecanismos más intuitivos a la hora de representar un comportamiento, entendido como el conjunto de acciones y la forma en que estas se desencadenan a medida que diversos eventos van sucediendo, es mediante máquinas de estados. Los diagramas de estados [Har87], son una herramienta poderosa para describir la lógica interna de un

sistema. Es posible definir aspectos complejos como la jerarquía y la concurrencia usando un conjunto reducido de elementos como son:

- **estado** : Situación particular en la que se encuentra un sistema.
- **evento** : Suceso externo que puede modificar el estado de un sistema.
- **condiciones de guarda** : Condiciones que se deben cumplir para permitir el cambio de estado.
- **transición** : Relación entre dos estados. Indica cómo evoluciona un sistema cuando se recibe un evento y se cumplen las condiciones de guarda.

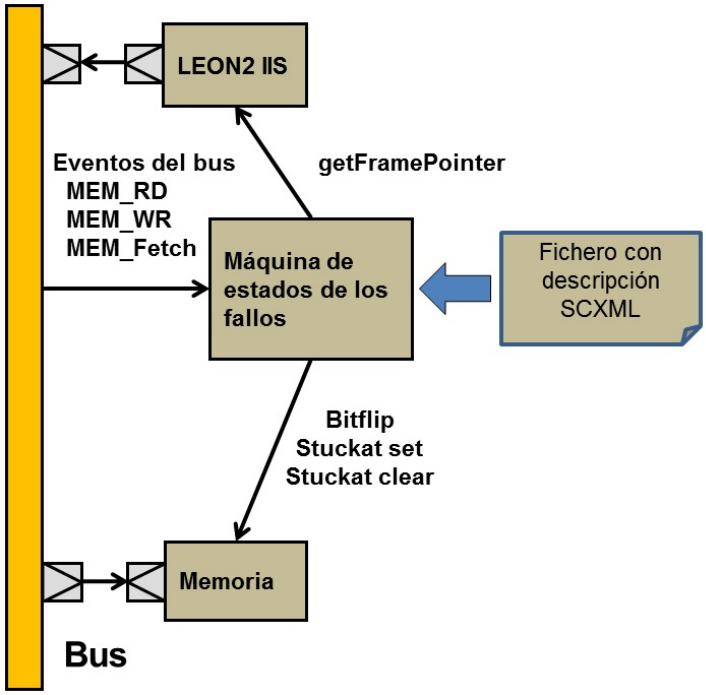


Figura 3.11: Integración de la máquina de estados SCXML con "Leon2ViP"

Como se describe en la figura 3.11 se ha dotado a "Leon2ViP" de un mecanismo que permita especificar la activación y desactivación de las diferentes condiciones de fallos mediante máquinas de estado. Los eventos de entrada serán los eventos de bus y en respuesta a estos eventos y las condiciones de guarda impuestos se procederá a la activación/desactivación de los fallos mediante los comandos proporcionados al efecto por el simulador. Dicha dinámica será expresada mediante el lenguaje de marcas propuesto por el W3C para la especificación de máquina de estados dirigidas por eventos State Chart XML (SCXML), sección 3.6.3.1

3.6.3.1. Descripción de máquinas de estados mediante SCXML

SCXML [CRM+14] es un especificación que proporciona un mecanimsmo sencillo para la descripción de máquinas de estados mediante un fichero XML. Una vez escrita la especificación, esta debe ser adecuadamente interpretada, para este fin se ha usado SSCXML (Simple SCXML) [SID14], una librería C que proporciona una implementación de referencia de un procesador SCXML.

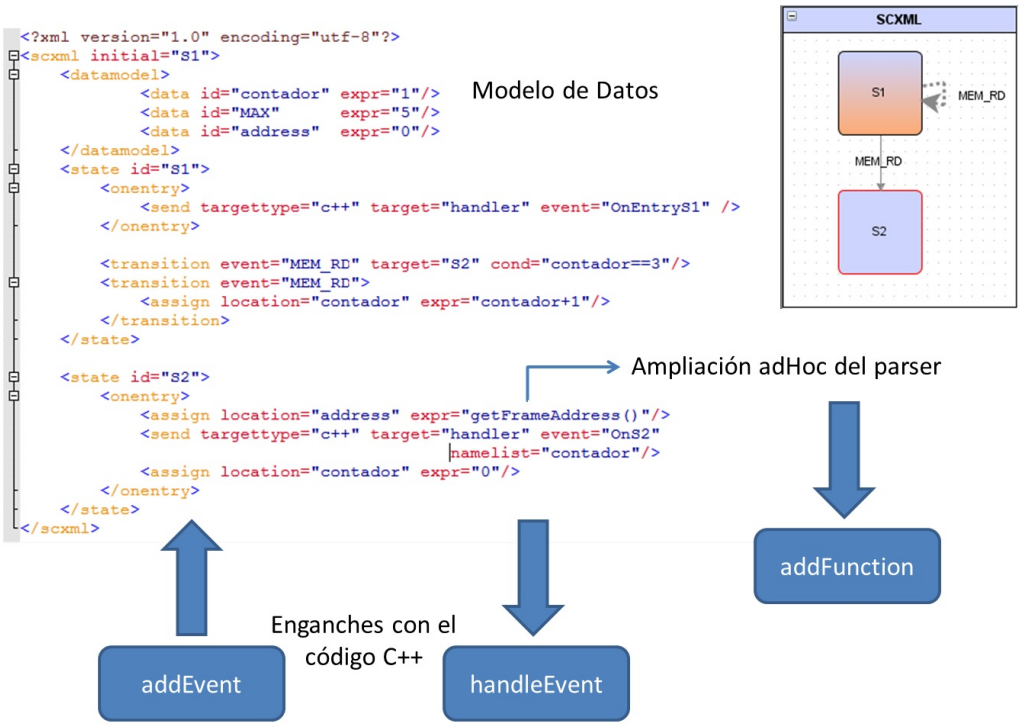


Figura 3.12: Ejemplo SCXML y enganches con el código del simulador

La figura 3.12 muestra el ejemplo de una máquina de estados y el modelo de datos asociado a la misma. La máquina consta de dos estado denominados "S1" y "S2". "S1" es el estado inicial y para cada evento de entrada "MEM_RD" incrementa un contador. Cuando se alcanza un máximo se pasa al estado "S2". En la misma figura se muestran los enganches con el código C++:

- addEvent : Permite la inserción de eventos en la máquina de estados. Un evento consiste un nombre y una lista de parejas variable-valor.
- handleEvent : Mediante la etiqueta <send> se envían eventos desde la máquina de estados al procesador SCXML "Leon2ViP".

```
bool SCXML_processor::handleEvent( const SSCXML::ScxmlString& _senderID ,
                                   const SSCXML::ScxmlString& _event,
                                   const SSCXML::VALUEMAP*    mapa,
                                   const SSCXML::ScxmlString& hints ) {
     // Procesamiento de los eventos y comunicación con "Leon2Vip"
}
```

- addFunction : Mecanismo de ampliación de las capacidades básicas del procesador. En el ejemplo se invoca la función "getFrameAddress" cuyo comportamiento se añade al procesador SCXML.

```
SSCXML::IValue* getFrameAddress( SSCXML::IStateMachine *_sm,
                                 const SSCXML::VALUEARRAY* _valList) {
     // procesamiento de la llamada
}

SSCXML::IParser *parser = proc.m_stateMachine->getExpressionParser();
parser->addFunction("getFrameAddress", &getFrameAddress);
```

Mediante este esquema es muy sencillo describir condiciones de disparo complejas para las campañas de fallos así como describir escenarios de fallos intermitentes. Además, aunque los ficheros SCXML pueden ser escritos mediante un sencillo editor de textos existen editores gráficos de libre distribución que pueden ser usados para la generación de los mismos.

3.6.3.2. Inyección de fallos sobre variables locales de funciones

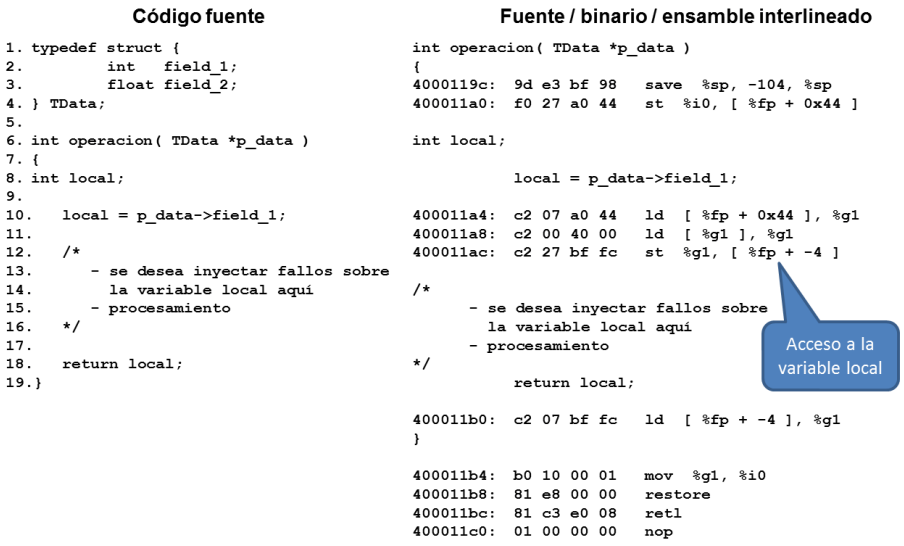


Figura 3.13: Localización de variables locales en el mapa de memoria

En este ejemplo se describe el procedimiento seguido para realizar una inyección de fallos sobre una variable local de una función. Este tipo de variables no tienen una ubicación absoluta en el mapa de memoria sino que ocupan una posición relativa en la pila de llamada a la función. La pila ocupa posiciones diferentes dependiendo del punto de invocación de la función. La figura 3.13 muestra el código fuente y un desensamblado del binario correspondiente a la función. Concretamente la variable ocupa un desplazamiento de -4 posiciones respecto al marco de la pila, `fp + -4`. En cuanto a las condiciones de activación/desactivación del fallo se desea lo siguiente:

- La activación del fallo se producirá en el tercer acceso en lectura a la variable y durante la segunda invocación de la función.
- La desactivación del fallo se producirá en el cuarto acceso en lectura a partir de la activación.

Estas condiciones solo pretenden reflejar las capacidades del sistema en cuanto a la defición del instante de fallo y la duración del mismo. Con las condiciones establecidas el modelo de datos necesario es el mostrado en la figura 3.14.

```
<datamodel>
        <data id="NumFetch"         expr="2"/>     <!-- Número de invocaciones                    -->
        <data id="CntFetch"         expr="0"/>     <!-- Contador de accesos a la función          -->
        <data id="FetchAddress"     expr="1000"/>  <!-- Dirección de comienzo de la función       -->
        <data id="CurrentAddress"   expr="0"/>     <!-- Dirección actual                          -->
        <data id="FrameAddress"     expr="0"/>     <!-- Marco de pila de la llamada actual        -->
        <data id="LocalOffset"      expr="-4"/>    <!-- Desplazamiento de la variable local       -->
        <data id="NumAccess_on"     expr="3"/>     <!-- Número de accesos para activar el fallo   -->
        <data id="CntAccess_on"     expr="0"/>     <!-- Contador de accesos para la activación    -->
        <data id="Stuckat1_mask"    expr="1"/>     <!-- Mascara a aplicar                         -->
        <data id="NumAccess_off"    expr="4"/>     <!-- Número de accesos para desactivar el fallo -->
        <data id="CntAccess_off"    expr="0"/>     <!-- Contador de accesos para la desactivación  -->
</datamodel>
```

Figura 3.14: Modelo de datos para una inyección sobre variable local

La figura 3.15 describe la máquina de estados que gobierna la inyección del fallo. Consta de tres estados:

- S1 : Recibe los eventos MEM_FETCH y cada vez que la dirección actual coincide con la función incrementa el contador "CntFetch". Cuando se alcanza el valor máximo establecido por "NumFetch" se pasa al estado "S2".
- S2 : En la entrada al estado obtiene el valor actual del marco de pila mediante "getFrameAddress". Para cada acceso en lectura a la variable se incrementa "CntAccess_on". Cuando se alcanza el máximo establecido en "NumAccess_on" se pasa al estado "S3".
- S3 : En la entrada al estado se envía mediante la etiqueta <send> la orden "stuckat_1" con los parámetros correspondientes para que el procesador active el fallo. Para cada acceso en lectura a la variable se incrementa "CntAccess_off". Cuando se alcanza el máximo establecido en "NumAccess_off" se pasa al estado "S4".
- S4 : Este es el estado final. Envía la orden "stuckend" al procesador y la ejecución proseguiría sin fallos.

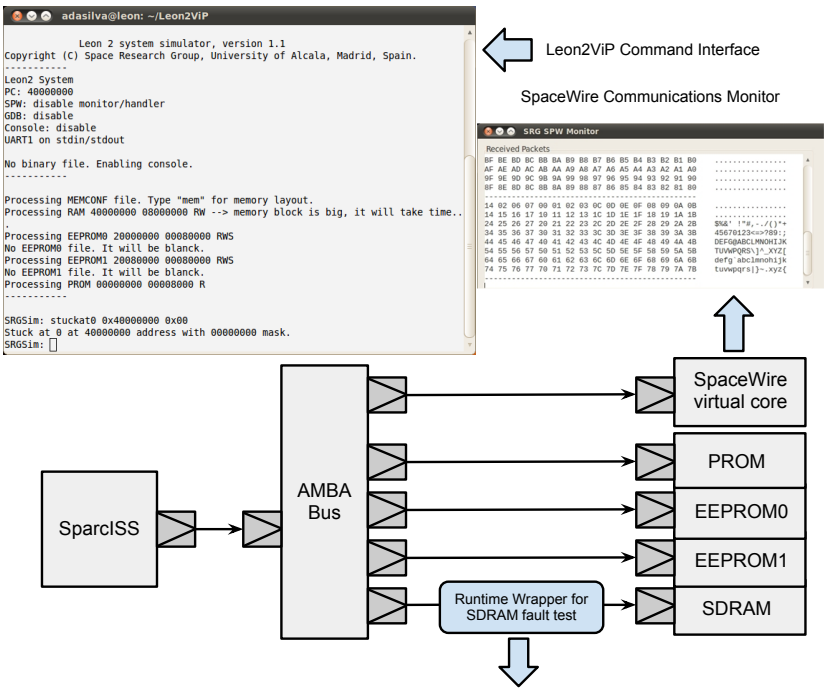
```
<?xml version="1.0" encoding="utf-8"?>
<scxml initial="S1">
    <datamodel>...</datamodel>

    <state id="S1"> <!-- Cuenta invocaciones de la función -->
        <transition event="MEM_FETCH" target="S2" cond="CntFetch==NumFetch"/>
        <transition event="MEM_FETCH">
            <if cond="CurrentAddress==FetchAddress">
                <assign location="CntFetch" expr="CntFetch+1"/>
            </if>
        </transition>
    </state>

    <state id="S2"> <!-- Cuenta accesos a la variable local para activar el fallo-->
        <onentry>
            <assign location="FrameAddress" expr="getFrameAddress()"/>
        </onentry>
        <transition event="MEM_RD" target="S3" cond="CntAccess_on==NumAccess_on"/>
        <transition event="MEM_RD">
            <if cond="CurrentAddress==(FrameAddress+LocalOffset)">
                <assign location="CntAccess_on" expr="CntAccess_on+1"/>
            </if>
        </transition>
    </state>

    <state id="S3"> <!-- Cuenta accesos a la variable local para mantener el fallo -->
        <onentry>
            <send targettype="c++" target="handler" event="stuckat_1"
                                                    namelist="CurrentAddress, Stuckat1_mask"/>
        </onentry>
        <transition event="MEM_RD" target="S4" cond="CntAccess_off==NumAccess_off"/>
        <transition event="MEM_RD">
            <if cond="CurrentAddress==FrameAddress+LocalOffset">
                <assign location="CntAccess_off" expr="CntAccess_off+1"/>
            </if>
        </transition>
    </state>

    <final id="S4"> <!-- Desactivación del fallo -->
        <onentry>
            <send targettype="c++" target="handler" event="stuckend" namelist="CurrentAddress"/>
        </onentry>
    </final>
</scxml>
```

Figura 3.15: Máquina de estados para una inyección sobre una variable local

3.7. Inserción de *wrappers* en el modelo "Leon2ViP"

Para la inyección de fallos transitorios se accede directamente al componente, en este caso la memoria, y se realiza la modificación solicitada. Sin embargo, los fallos temporales y/o permanentes necesitan una infraestructura que impida la actualización de los datos en memoria. Para ello se procede a la inserción dinámica de *wrappers* entre los componentes del modelo mediante la técnica descrita en la sección 2.5.1, concretamente entre el bus y la memoria, aplicando una máscara en todos los accesos que se realicen a las direcciones indicadas en la ventana de órdenes. Una descripción detallada de la técnica utilizada para descripciones TLM2 se puede encontrar en los apéndices A.1 y A.4 [dSS09b, dSPPS].

La figura 3.16 detalla la inserción real de los *wrappers* en el modelo "Leon2ViP", en el ejemplo entre el bus y el módulo SDRAM.




```
void tlm2_parser_wrapper::b_transport_processing( transaction_type& trans, sc_core::sc_time& delay )
{
   // cout << "---B_TRANSPORT Wrapping---" << endl;

   // if WRITE command and address match, apply MASK on data that is going to be written

   // Call target original b_transport  on SDRAM
   b_transport_original_path_call( trans, delay );

   // if READ command and address match, apply MASK on data read
}
```

Figura 3.16: Inserción de la instrumentación

Capítulo 4

Resultados de la investigación

Sheldon Cooper: «No estoy loco. Mi madre me hizo pruebas»

The Big Bang Theory

Agente Smith : «Nunca envíes a un humano a hacer el trabajo de una máquina»

The Matrix

4.1. Introducción

Para el desarrollo del software de arranque se ha seguido un enfoque basado en componentes empleando el entorno de desarrollo MICOBS [Esp12], desarrollado por el propio grupo SRG. De acuerdo con este enfoque, la aplicación está dividida en módulos o paquetes software reutilizables (*software packages*). Cada uno de estos paquetes se encuentra almacenado y versionado en un repositorio específico. Además del propio código de los paquetes, en el repositorio se almacenan distintos meta-datos como, por ejemplo, los relativos a las propiedades extra-funcionales del paquete.

Una de las principales características del entorno de trabajo MICOBS es que incluye un modelo de plataforma de despliegue (*deployment platform*). De acuerdo con este modelo se pueden definir las distintas combinaciones tanto hardware (microprocesador, placa, etc.) como software (sistema operativo) sobre las que se pueden ejecutar las aplicaciones empotradas. Una vez definidos los paquetes, la aplicación se construye mediante otro modelo denominado proyecto de despliegue (*deployment project*). En este proyecto, se establece, además del conjunto de paquetes software que forman la aplicación, las distintas plataformas sobre las que dichos paquetes se pueden desplegar. En función del tipo de proyecto que se trate, estas plataformas pueden ser, por ejemplo, los distintos modelos hardware que se definen durante las etapas de desarrollo (*breadboard*, de ingeniería, de calificación, etc.) o plataformas simuladas, como "Leon2ViP". Una vez definido el modelo de despliegue y seleccionado una plataforma objetivo, MICOBS permite generar de forma automática los archivos necesarios para configurar la aplicación y construir la imagen ejecutable final. En el caso del software de arranque, se ha definido un proyecto de despliegue general de la aplicación completa y una serie de proyectos adicionales para realizar las

pruebas unitarias. En el caso de las pruebas unitarias, los proyectos emplean tres plataformas de despliegue distintas: una basada en Linux sobre una arquitectura intel de 32 bits y otras dos basadas en el sistema operativo RTEMS 4.8 ejecutándose bien sobre la placa SRG A3P o bien sobre el simulador "Leon2ViP".

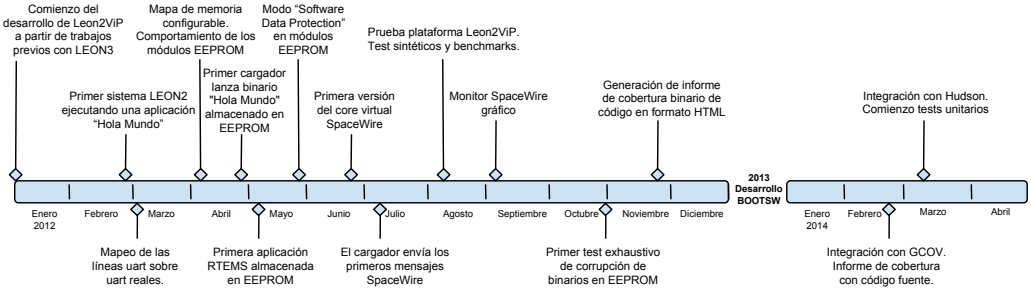


Figura 4.1: Desarrollo de "Leon2ViP"

La figura 4.1 muestra el tiempo de desarollo de "Leon2ViP", donde se enumeran los hitos principales de desarrollo de la plataforma virtual y de las herramientas asociadas como son el monitor de SpaceWire, la visualización de la cobertura de código y el uso de herramientas de integración continua, etapas que se describen más adelante.

4.1.1. Escenarios de prueba contemplados con "Leon2ViP"

En este apartado se describen los diferentes escenarios en los que se ha usado la plataforma virtual "Leon2ViP" dentro del desarrollo y prueba del software de *boot* de la ICU. Hay que distinguir dos casos:

1. Escenarios en los que se prueba la interacción del BOOTSW con otros componentes de la nave a través de la interfaz SpaceWire con presencia de fallos en memoria o en las comunicaciones.

2. Escenarios de prueba funcionales del BOOTSW con análisis de cobertura del código.

Todo el proceso de arranque de la ICU debe ser cuidadosamente verificado, de forma que el BOOTSW funcione correctamente en caso de ser necesario, ya que su actualización no es posible una vez lanzado *Solar Orbiter*. Uno de los requistos exigidos en las pruebas es conseguir una cobertura del 100 % del código en lo que se refiere a *statement coverage*[1] y *conditional coverage*[2]. Esto significa que todo el código que gestiona los diferentes escenarios de fallo considerados en las especificaciones debe ser ejercitado para verificar que su comportamiento se ajusta a lo esperado. Para el primer caso y de acuerdo con los mecanismos de *Fault Detection Isolation and Recovery* (FDIR) definidos para el BOOTSW, este debe ser capaz:

[1] Líneas de código ejecutadas
[2] Expresiones condicionales evaluadas

- Detectar la corrupción debida a SELs en áreas de memoria pertenecientes a bloques EEPROM y SDRAM, usadas por los binarios de aplicación, y reportar el estado al ordenador central.
- Recibir y decodificar adecuadamente en presencia de errores, los telecomandos de actuación recibidos.
- Proceder al regrabado de los bancos EEPROM con los parches de memoria recibidos y que evitan las zonas dañadas.

Para el segundo caso, la plataforma virtual "Leon2ViP" debe integrarse con las herramientas GNU usadas para el análisis de la cobertura y con el sistema de integración continua que gestiona la construcción del binario en función de las actualizaciones del código fuente y la correcta verificación de los tests que se han definido.

4.2. Verificación de la secuencia de arranque

Se ha llevado a cabo una campaña de fallos exhaustiva del software de arranque. En esta campaña se ha realizado un barrido con fallos *stuckat* de todas las posiciones de memoria correspondientes a los binarios almacenados en EEPROM y en las zonas de despliegue de los mismos, correspondientes a SDRAM. Los test realizados corresponden a siete posibles escenarios:

- Sin corrupción de memoria durante el arranque.
- Corrupción de la versión *Baseline* almacenada en el primer banco EEPROM. La zona de despliegue SDRAM funciona correctamente.
- Corrupción de la versión *Updatable* almacenada en el segundo banco EEPROM. La zona de despliegue SDRAM funciona correctamente.
- Corrupción de las dos versiones del binario de aplicación. Las zonas de despliegue SDRAM funcionan correctamente.
- Corrupción de la zona de despligue SDRAM correspondiente al binario de aplicación *Baseline*. Los valores almacenados en EEPROMs son correctos.
- Corrupción de la zona de despligue SDRAM correspondiente al binario de aplicación *Updatable*. Los valores almacenados en EEPROMs son correctos.
- Corrupción de las zonas de despliegue SDRAM correspondientes a las dos versiones del binario de aplicación. Los valores almacenados en EEPROMs son correctos.

Para cada uno de los escenarios descritos previamente, el comportamiento del BOOTSW puede ser:

- Arranque de la versión *Baseline*.
- Arranque de la versión *Updatable*.

Tabla 4.1: Escenarios de arranque de la ICU

		COMPORTAMIENTO			
Fallos SDRAM	Baseline y Updatable NOK	Sin boot TM: 54	Sin boot TM: 54	Sin boot TM: 54	Sin boot TM: 54
	Updatable NOK	Baseline boot TM: 53, 51	Sin boot TM: 54	Baseline boot TM: 53, 51	Sin boot TM: 54
	Baseline NOK	Updatable boot TM: 53, 51	Updatable boot TM: 53, 51	Sin boot TM: 54	Sin boot TM: 54
	SDRAM OK	Updatable boot TM: 51	Updatable boot TM: 53, 51	Baseline boot TM: 53, 51	Sin boot TM: 54
		EEPROM OK	Baseline NOK	Updatable NOK	Baseline y Updatable NOK
		Fallos EEPROM			

- No se puede arrancar la aplicación, envío de mensaje de telemetría a la nave indicando la situación y espera de telecomandos.

En la tabla 4.1 se resumen los diferentes escenarios con fallo y el comportamiento esperado del BOOTSW. Los mensajes de telemetría enviados se corresponden con un servicio tipo 5 (*Event reporting*). El mensaje TM(51) es un informe nominal sin errrores, TM(53) es un informe de severidad media y TM(54) es un informe de severidad alta.

El número total de arranques del sistema, necesarios para comprobar los seis escenarios de error definidos previamente, depende del tamaño del binario de aplicación almacenado en los bancos de memoria EEPROM. Por ejemplo para un binario de 256kbyte se tendrían 262144 * 6 = 1.572.864 posibles arranques. El tiempo medio empleado en la configuración de un escenario, arranque de la plataforma virtual y dentro de la plataforma, el arranque del BOOTSW se sitúa entre 2 y 3 segundos. Ello implica un periodo de prueba de alrededor de 50 días en una única máquina. Este tiempo puede reducirse de forma significativa puesto que la plataforma virtual puede ejecutarse paralelamente en varios equipos sin grandes requisitos en cuanto a prestaciones, repartiendo los escenarios de test entre todos ellos.

Para cada uno de los escenarios de prueba se genera un informe de cobertura en HTML donde el ingeniero de pruebas puede comprobar la porción de código ejecutada. Esta información es usada para la generación de nuevas campañas de inyección de fallos que fuercen la cobertura del código restante. La figura 4.2 muestra un ejemplo de la apariencia del informe HTML generado, donde se representan en diferentes colores las porciones de código que han sido ejercitadas y el número de veces que el procesador ha ejecutado una determina instrucción. Este esquema de cobertura del binario es útil para las partes del software que han sido codificadas en lenguaje de ensamble y se complementa con el procedimiento implementado para la cobertura del código fuente, véase la sección 4.5.

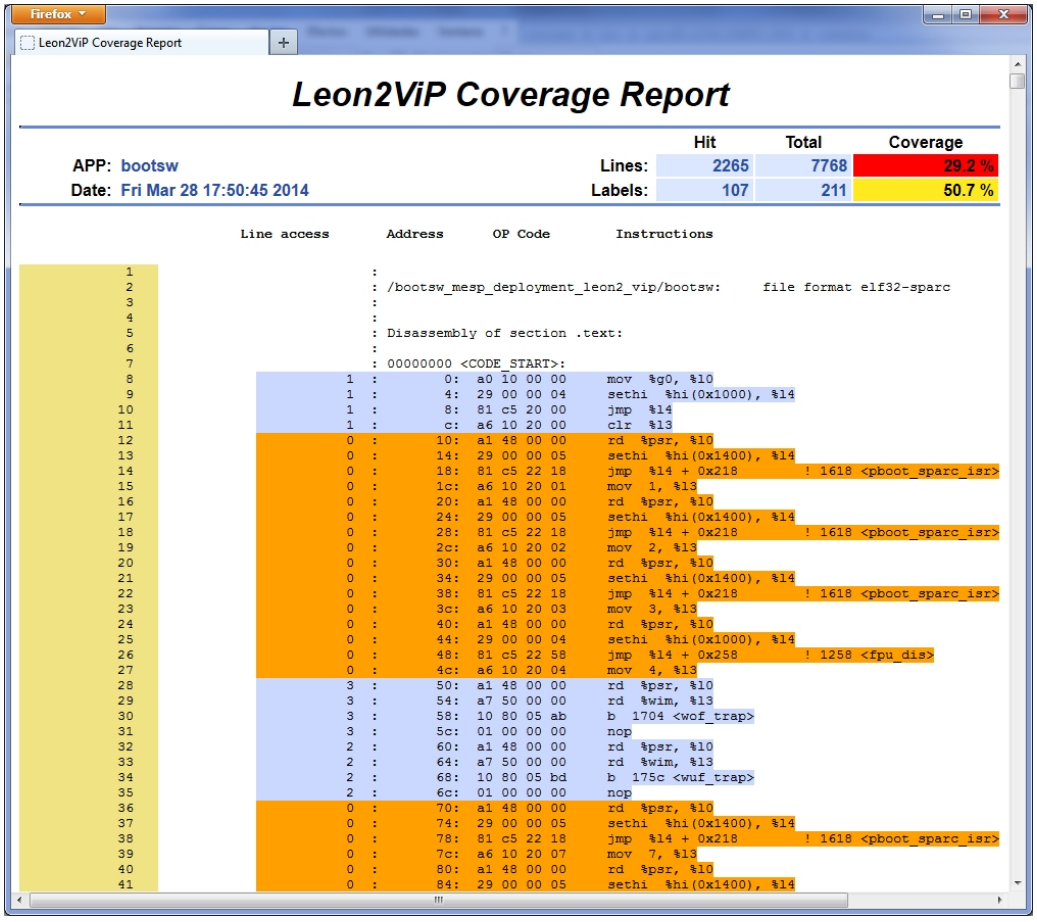


Figura 4.2: Ejemplo de informe de cobertura de una prueba generado por "Leon2ViP"

4.3. Verificación del intérprete de telecomandos

La ICU se comunica con el ordenador central de la nave mediante una línea SpaceWire. Mediante esta interfaz de comunicaciones, el BOOTSW informa del estado operativo de la ICU y recibe las correspondientes órdenes por parte del ordenador central. De acuerdo con la especificación, la ICU debe generar un mensaje de telemetría tipo 1 (*Telecommand verification*), indicando si ha aceptado o no el telecomando enviado previamente. La verificación de la implementación del proceso de decodificación de los paquetes es esencial y necesita del envío de paquetes mal formados. Mediante el uso del monitor de SpaceWire es posible la alteración de campos de los paquetes enviados al BOOTSW y realizar las comprobaciones siguientes:

- Identificación de destinatario correcto, el paquete debe ir destinado a EPD.

- Parámetros de la cabecera del telecomando correctos.
- Longitud de paquete coincidente con el campo longitud de la cabecera.
- Cálculo el CRC del paquete correcto.
- Verificación de la validez del tipo y subtipo de paquete recibido.
- Cabecera *Packet Utilisation Standard* (PUS) correcta.

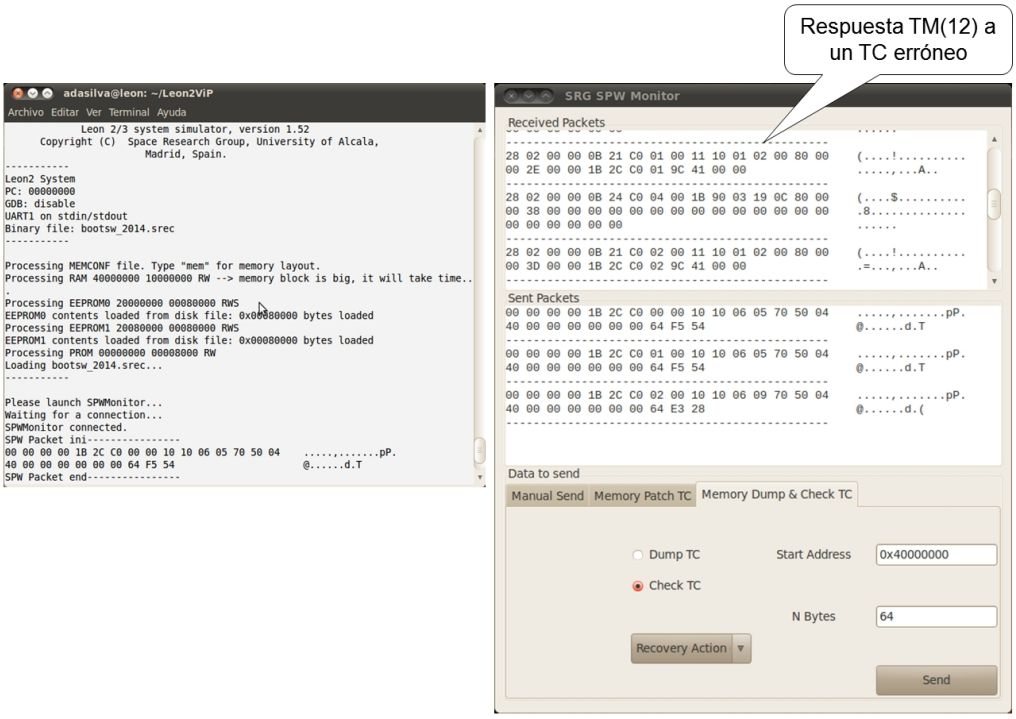


Figura 4.3: Prueba del intérprete de telecomandos

Para cada una de las situaciones anteriores, se han generado paquetes erróneos mediante el monitor de SpaceWire. Estos paquetes se han enviado al BOOTSW y se ha comprobado que los mensajes de telemetría retornados se corresponden con un mensaje TM(12) *TC Acceptance Report*. En la figura 4.3 se muestra como ejemplo un caso de prueba con el código de retorno para un telecomando erróneo.

4.4. Verificación del proceso de actualización remota de los binarios en EEPROM

La plataforma virtual "Leon2ViP" emula el comportamiento de los mecanismos SDP presentes en las memorias EEPROM. Inicialmente la escritura está prohibida y es nece-

sario realizar una secuencia de escritura de datos en direcciones concretas que habiliten la capacidad de escritura en la memoria. Por otro lado, el número de veces que se puede proceder a la la reescritura de datos en memorias EEPROM reales está limitado. Aunque esta cantidad es más que suficiente para el tiempo de operación de la ICU, es una buena práctica realizar el desarrollo y las pruebas en el entorno virtual y la comprobación definitiva, con una versión madura del procedimiento, sobre el hardware real.

Para cada uno de los escenarios descritos en la tabla 4.1 en los que no se puede realizar el *boot* de la ICU, el BOOTSW pasa a un estado denominado *Safe Mode* donde espera telecomandos de servicio de tipo 6 (*Memory Management*). Este tipo de telecomandos permite operaciones de *PATCH*, *DUMP* y *CHECK* de la memoria. Mediante la interfaz virtual de SpaceWire descrita en la sección 3.5 es posible emular la comunicación con el ordenador central de la nave y someter al BOOTSW a diversos escenarios de prueba. De acuerdo con la especificación del formato de telecomandos tipo 6 (Gestión de memoria), el tamaño máximo del campo de datos de una operación *PATCH* es de 226 bytes. Esto significa que son necesarios alrededor de 145 (0x8000 / 226) mensajes para enviar la imagen nueva completa de una EEPROM. Para las pruebas se ha partido de un escenario inicial con los dos bancos EEPROM en blanco. Este caso se correspondería con la situación *Baseline y Updatable NOK* de la tabla 4.1. En una primera fase se ha procedido al envío de la versión *Updatable* y se ha comprobado el correcto grabado del binario, con el consiguiente arranque normal de la aplicación y la recepción del mensaje de telemetría asociado. En una segunda fase se ha procedido al envío de telecomandos para el grabado de la versión *Baseline*. Finalmente el sistema ha arrancado con el envío del mensaje de telemetría TM(51) correspondiente a un informe nominal sin errores.

4.5. Integración de "Leon2ViP"con GCOV para el análisis de cobertura del código fuente

Las pruebas de cobertura del software de arranque de la ICU se han dividido en dos niveles: cobertura de código máquina y de código en lenguaje C. Para realizar la cobertura del binario se emplea la funcionalidad de cobertura de código objeto de "Leon2ViP" tal como se muestra la sección 4.2. Sin embargo en este informe, aparte de los símbolos globales como los nombres de las rutinas, no hay información que relacione el binario con el código fuente escrito en un lenguaje de alto nivel. Puesto que las herramientas de compilación usadas son GNU, se ha procedido a la integración de la herramienta de análisis de cobertura GCOV en el entorno de "Leon2ViP". La figura 4.4 muestra el escenario general de uso que consta básicamente de tres fases:

- Fase de compilación: En esta fase se generan los binarios. Mediante el uso de *switchs* de compilación específicos el compilador añade la instrumentación de código adecuada. Esta instrumentación se encarga de ir incrementando una serie de contadores que se corresponden con los bloques de código C de la aplicación. Así, los contadores reflejan la cantidad de veces que se ha ejecutado cada uno de dichos bloques. El código necesario para la instrumentación se encuentra en una librería del compilador denominada LIBGCOV. Además, por cada fichero fuente se genera un fichero

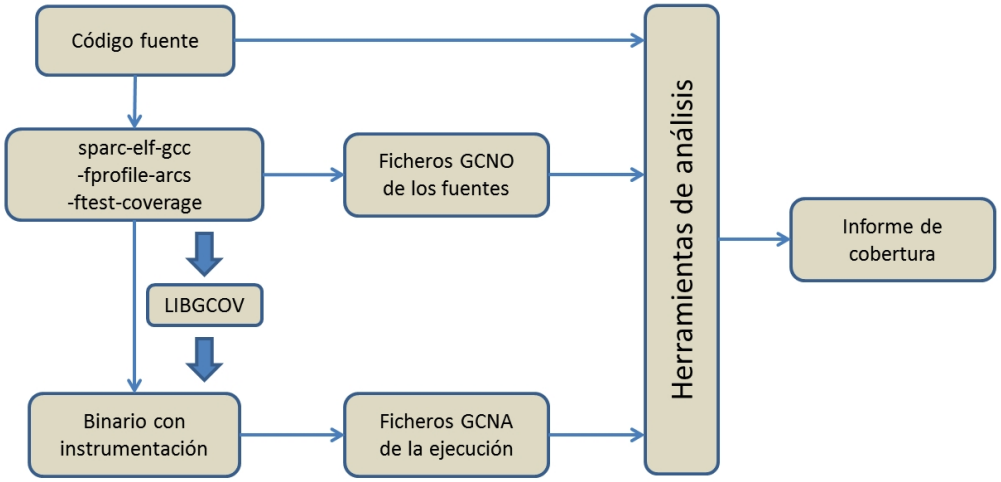


Figura 4.4: Análisis de cobertura con GCOV

GCNO que contiene la información necesaria para poder recuperar a partir de los bloques de código binario, las líneas de código fuente asociadas.

- Fase de recolección: En esta fase se ejecuta el programa con la instrumentación añadida y se procede al almacenamiento de la información de cobertura en estructuras de datos internas. Finalizada la ejecución del programa, la información de cobertura se almacena en ficheros GCNA. Se genera un fichero GCNA para cada módulo fuente.
- Fase de análisis: A partir de la información estática, ficheros fuente y GCNO, y de la información obtenida en tiempo de ejecución se generan los informes de cobertura.

Este entorno está pensado para la prueba y verificación de software en un entorno que disponga de soporte para un sistema de ficheros. Este hecho dificulta su uso para el análisis en sistemas embebidos sencillos ya que no es posible obtener de forma sencilla la información de los ficheros GCNA. Concretamente en la ICU no hay sistema de ficheros. Sin embargo, puesto que el código fuente de la librería LIBGCOV está disponible, es posible modificarlo para adaparlo a un escenario específico. Prácticamente todo el código de LIBGCOV es C puro y solo necesita servicios del sistema operativo subyacente, al final de la ejecución del programa, cuando desea guardar los datos obtenidos durante la ejecución del programa. Esto se realiza en la función *_gcov_exit*, la cual en la distribución estándar escribe los datos en el sistema de ficheros. Modificando esta función es posible redirigir la escritura de los datos a otro lugar o periférico.

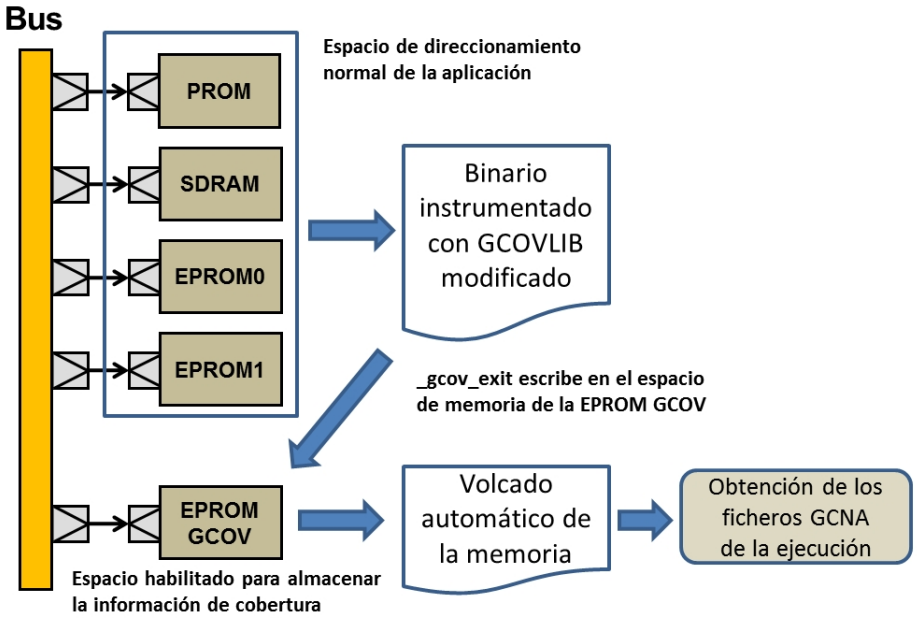


Figura 4.5: Adaptación de LIBGCOV a "Leon2ViP"

La figura 4.5 muestra el escenario implementado para dar soporte GCOV en la plataforma virtual "Leon2ViP".

- Se ha modificado *_gcov_exit* para que realice el volcado de los datos en un bloque de memoria en lugar de escribirlos en el disco y se ha generado una nueva LIBGCOV para el compilador *sparc-elf-gcc*.

- Haciendo uso del fichero de configuración del mapa de memoria de "Leon2ViP" se define un nuevo bloque de memoria EEPROM (*EEPROM GCOV*), fuera del espacio de direccionamiento normal del BOOTSW. En este espacio de memoria se escribe la información de cobertura del programa ejecutado.

- "Leon2ViP" vuelca el contenido de esta memoria a un fichero externo al finalizar la ejecución. De este volcado se extraen los ficheros GCNA que junto con los GCNO obtenidos en la fase de compilación y los ficheros fuente permiten la elaboración de informe de cobertura en el formato que se estime oportuno.

En la figura 4.6 se muestra un ejemplo de informe de cobertura de código fuente generado después de la ejecución de un test.

Figura 4.6: Ejemplo de informe de cobertura generado con el soporte GCOV

4.6. Integración de "Leon2ViP"en un sistema de integración continua

La integración continua tiene como objetivo principal verificar que el binario generado con las actualizaciones del código fuente realizadas por los desarrolladores, sigue funcionando correctamente con todos los test creados anteriormente. De esta forma se detectan los posibles errores en una fase muy temprana, identificando más fácilmente los cambios que han provocado algún tipo de malfuncionamiento, haciéndose más sencilla la corrección de los problemas detectados. Para ello, el servidor una vez por semana descarga el código de todos los proyectos correspondientes a las pruebas unitarias, lo compila y lo ejecuta ejecuta para la plataforma de despliegue basada en Linux y la basada en la plataforma virtual "Leon2ViP". En la figura 4.7 se observa la situación de la plataforma virtual en el proceso global de generación y prueba del BOOTSW.

Para poder integrar la plataforma "Leon2ViP" en un sistema de integración continua es necesario poder automatizar la configuración y ejecución de un programa. Como se comentó en la sección 3.2, "Leon2ViP" dispone de tres modos de funcionamiento:

- Modo de ejecución *standalone*: En este modo, mediante parámetros de línea de comandos se especifica a la plataforma el fichero con el código que hay que ejecutar y la dirección de comienzo del contador de programa. LeonViP carga el binario y lanza la ejecución en la dirección especificada. El programa se ejecuta sin interrupción hasta su finalización.

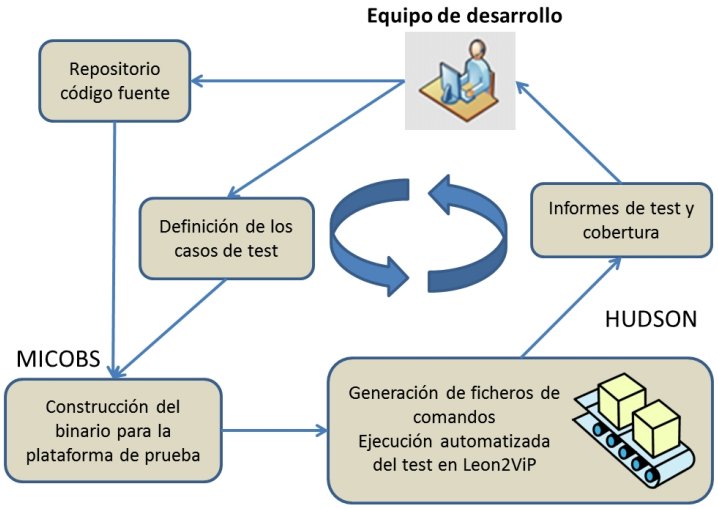


Figura 4.7: Integración de "Leon2ViP" en el sistema de integración continua

- Modo normal con consola: La plataforma arranca normalmente y ofrece una consola de comandos al usuario. Además de cargar el binario externo que se desee, en este escenario es posible la definición de puntos de ruptura, ejecución del programa *paso a paso*, inyección de fallos y la inspección del contenido de la memoria y de los registros.
- Modo *batch* mediante fichero de comandos: Todos los comandos disponibles para el usuario en modo consola, pueden ser escritos en un fichero de comandos que será ejecutado por parte de "LeonViP" cuando se arranca en modo *batch*. Este mecanismo ofrece la posibilidad de automatizar la realización de una batería de pruebas. Para cada prueba que se desee, un agente externo puede generar el fichero de comandos, lanzar la ejecución y recoger los resultados.

Como plataforma de integración continua se ha usado la herramienta Hudson [HUS14]. Esta herramienta de libre distribución se ha instalado sobre un PC ejecutando un entorno basado en la distribución de Linux Ubuntu Server. Para facilitar su manejo, Hudson proporciona un servidor web que permite definir los trabajos que ha de realizar y controlar el resultado de su ejecución. El servidor genera una serie de informes del resultado de todo el proceso. Entre los informes generados, de la ejecución de las pruebas sobre "Leon2ViP", el servidor obtiene un informe de cobertura siguiendo un proceso idéntico al mencionado en el apartado anterior. En lugar de utilizar la extensión *lcov*, Hudson emplea una extensión específica que consigue integrar, dentro del propio servidor web, el informe resultante. La figura 4.8 muestra un informe de cobertura integrado en la herramienta. En dicho informe se desglosa la cobertura de cada paquete de código y de las funciones que lo integran. Otro aspecto importante es el histórico de cobertura en función de las integraciones (*builds*) realizadas.

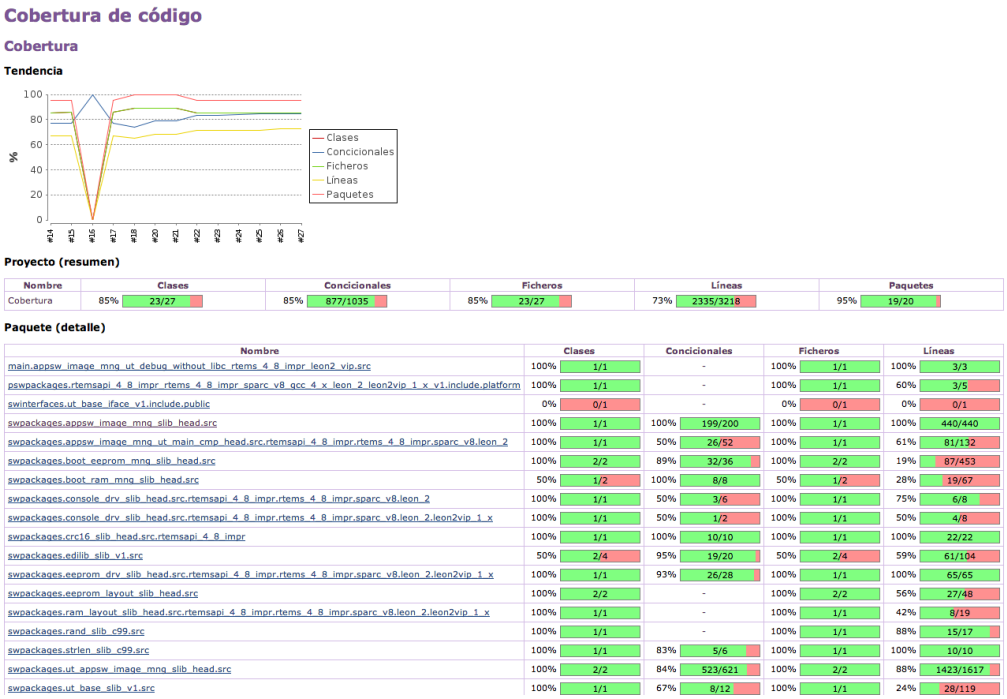


Figura 4.8: Informe de cobertura integrado en Hudson

4.6.1. Resumen de pruebas realizadas al software de arranque de la ICU

A la hora de escribir esta memoria, el proceso de escritura del BOOTSW y sus correspondientes test unitarios todavía no han concluido por lo que algunas de la cifras que se proporcionan a continuación, como pueden ser el número total de líneas de código (LOC) de los módulos del BOOTSW, así como de los test no son exactas, pero proporcionan una idea del esfuerzo dedicado a verificación. En total se han definido 18 test unitarios que prueban los diferentes módulos de los que se compone el BOOTSW. Cada uno de los test soporta un número variable de configuraciones con el fin de ejercitar todos los posibles flujos de ejecución presentes en el módulo que se está probando. En algunos casos esto solo es posible mediante inyección de fallos. Concretamente en 5 de los 18 test unitarios ha sido necesario definir una configuración que incluyera una campaña de fallos con el fin de alcanzar el 100 % de cobertura del código. Como ejemplo en la tabla 4.2 se muestran las métricas de uno de los módulos que componen el BOOTSW.

Tabla 4.2: Cifras de ejemplo del módulo *boot_utils*

Nombre del módulo	LOC módulo	Caminos (Branches)	LOC de test
boot_utils	199	86	254

La tabla 4.3 muestra las diferencias en cuanto a la cobertura de código del módulo sin y con inyección de fallos. Como se describe en la tabla, prácticamente la totalidad del código (93 %) es ejercitado mediante las configuraciones de test puramente funcionales, siendo necesaria la inyección de fallos para alcanzar el 100 %. Para el BOOTSW en su conjunto ha sido necesaria una configuración con inyección de fallos en casi un tercio, 5 de 18, de los test unitarios para alcanzar el 100 %.

Tabla 4.3: Evolución de la cobertura con inyección de fallos

	Sin inyección de fallos	Con inyección de fallos
LOC	185 (93 %)	199 (100 %)
Caminos	79 (92 %)	86 (100 %)

Capítulo 5

Conclusiones y líneas futuras

Hal 9000 : «El fallo debe ser atribuido sólo a un error humano. Esas cosas han sucedido más de una vez y siempre han sido debidas a error humano.»
2001, una odisea del espacio

Alicia : «¿Podría decirme, por favor, qué camino debería seguir desde aquí?»
Gato : «Eso depende en gran parte de a dónde quieras ir»
Alicia en el País de las Maravillas, Lewis Carroll

5.1. Introducción

Sin duda alguna, el uso de herramientas es imprescindible para manejar la complejidad de los desarrollos actuales. Sin embargo, el uso de entornos integrados, donde una gran parte de la toma de decisiones se han automatizado con la idea de evitar el fallo humano, pretendiendo así un aumento de la fiabilidad, se han encontrado con la denominada ironía de la automatización descrita en [Rea90]. En primer lugar, la automatización traslada la carga del diseño del desarrollador del sistema al diseñador de la herramienta, requiriendo que éste anticipe y trate correctamente todos los escenarios posibles; esta tarea parece antojarse imposible. En segundo lugar, los escenarios no contemplados automáticamente requieren la intervención humana; desgraciadamente estos escenarios excepcionales se corresponden con los aspectos más complejos y que requieren más *know how*.

Teniendo en cuenta lo heterogéneo de los equipos de desarrollo actuales, es normal que muchos directores de proyecto opten por buscar una única herramienta que integre todos los aspectos del desarrollo del sistema. Desde la exploración del espacio de diseño y el soporte al desarrollo temprano del software, hasta la verificación final del producto. Por supuesto, los fabricantes argumentarán que su herramienta es capaz de manejar todos los escenarios y una vez adquirida, el equipo se ve obligado a usarla independientemente de su adecuación al escenario, simplemente para justificar la compra. Este hecho es conocido como la paradoja del "martillo y el clavo", *si tu herramienta tiene forma de martillo, todos los problemas se tratarán como si fueran clavos.* Aunque en muchos casos esta aproximación pueda resultar correcta, no es la forma óptima de abordar el problema. Desde el punto de vista de la verificación software, se corre el riesgo de realizar solo aquellas

pruebas que la herramienta permite, ignorando otras que pueden ser de suma importancia para el dominio de aplicación concreto. La herramienta no debe dictar qué probar sino ser solo una baza más jugada en el proceso global de verificación del software embebido.

A pesar de todo hay que entender que no existe una "bala de plata" que asegure el correcto funcionamiento del software embebido en los sistemas actuales. Hacen falta herramientas, generalmente más de una, y personal cualificado que sepa usarlas adecuadamente. Aun así se siguen produciendo errores; por ejemplo, en fechas recientes diversas empresas automovilisticas han tenido que llamar a revisión a millones de unidades vendidas por problemas con el software embebido [EP14, Bub13].

5.2. Conclusiones

La capacidad de inyección de fallos en tiempo de ejecución de forma controlada y repetible, es imprescindible para la verificación de los mecanismos de mitigación incluidos en el software de vuelo de misiones espaciales. Además, la fase de verificación software debe comenzar lo antes posible para poder detectar posibles deficiencias en el comportamiento esperado del sistema y que no hayan sido avanzadas en la especificación del sistema. Con todo ello y con la experiencia adquirida en la realización de esta tesis, las conclusiones más importantes que se han obtenido han sido las siguientes:

- Es posible construir una plataforma virtual ad-hoc que dé soporte a las necesidades concretas de un proyecto, usando entornos abiertos como SystemC e interfaces de componentes interconectados en buses como TLM2.0. El uso de entornos comerciales cerrados en el entorno espacial es a día de hoy complicado debido al alto coste por unidad desarrollada.
- Se ha desarrollado una técnica específica de instrumentación dinámica de código en modelos TLM2.0. Esta técnica se ha usado para realizar inyección de fallos en la plataforma virtual pero puede ser usada con otros fines y es aplicable a la validación de componentes software de terceras partes, sin tener acceso al código fuente del componente.
- Se ha proporcionado un entorno de ejecución y depuración funcional del software de *boot*, de forma que su desarrolo pudiera comenzarse antes de tener una versión estable del hardware. Esto ha permitido el trabajo independiente de los equipos de desarrollo hardware y software con pocos problemas de concurrencia en el acceso a los recursos hardware. De esta forma el desarrollo del BOOTSW no se ha visto interrumpido en ningún momento por la indisponibilidad del hardware.
- Se ha proporcionado un entorno de inyección de fallos controlable y reproducible para el software de *boot*, imprescindible para poder ejercitar todos los mecanismos de excepción implementados en el software y lograr una cobertura del 100 % del código.
- La realimentación temprana de problemas hardware/software durante el desarrollo del software de *boot* y de IP cores específicos con sus correspondientes controladores de dispositvo, ha permitido una reducción importante en la brecha de comunicación entre los equipos de desarrollo hardware/software.

5.3. Líneas futuras de trabajo

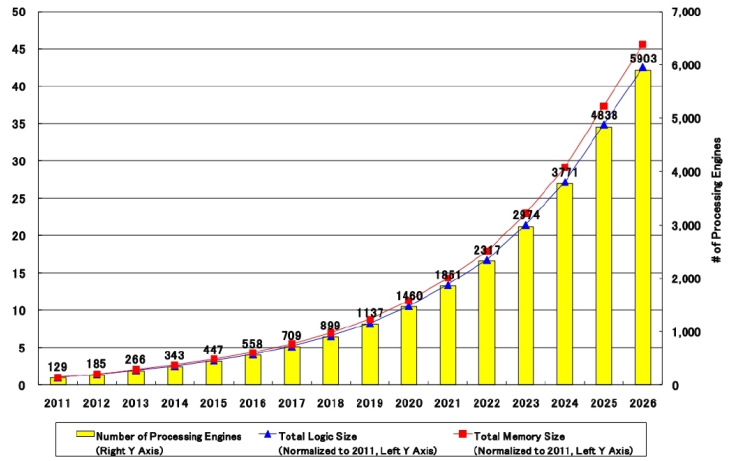


Figura 5.1: Tendencia en el número de procesadores en un único encapsulado [ITR11]

La complejidad de los diseños y el aumento del software parece ser hoy en día una tendencia imparable. De acuerdo con la organización ITRS, podrá haber 6000 procesadores integrados en un único encapsulado para 2026, véase la figura 5.1. Diseñar y verificar el software embebido en este tipo de sistemas representa un enorme reto. Entre los posibles desafíos se podrían concretar:

- Técnicas de modelado de sistemas monoprocesador y multiprocesadores para la evaluación de problemas de concurrencia y estimación *Worst Case Execution Time* (WCET): Por razones de consumo de potencia y tolerancia a fallos [Sci05], la frecuencia de reloj de los sistemas espaciales no es excesivamente alta de forma que hay espacio para aumentar la precisión del modelado del procesador o simular varios procesadores simultaneamente, manteniendo la cosimulación del software. En este contexto se pueden realizar estimaciones muy realistas de la capacidad de planificación del sistema. Aunque en el caso de sistemas de tiempo real estricto como la ICU, la memoria caché se encuentra deshabilitada, estas técnicas de modelado, de una forma general deberían incluir el modelado de las memorias caché con el fin de evaluar su efecto sobre la ejecución del software embebido, tanto en sistemas monoprocesador como multiprocesador [BCB+13, CGLH10, LGA+11].

- Estudio de las técnicas de virtualización aplicadas a sistemas de tiempo real y espaciales: La virtualización ha demostrado su efectividad en grandes sistemas y procesamiento de datos donde los tiempos de respuesta no son tan limitados como en un sistema de tiempo real. El uso en aplicaciones espaciales de procesadores virtualizados mediante el uso de hipervisores software con o sin soporte nativo del hardware y sus implicaciones en la planificación temporal de los sistemas virtualizados es un campo activo de investigación [MRCM09].

- Técnicas de simulación de vida acelerada del software: Muchos problemas software se manifiestan solamente después de un tiempo prolongado de actividad del software. Estas técnicas permitirían comprimir un tiempo de operación dilatado en un tiempo de prueba menor.

Bibliografía

[AAA+90] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: a methodology and some applications. *Software Engineering, IEEE Transactions on*, 16(2):166–182, Feb 1990.

[AE02] T. Austin and E. Larsonand D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, Feb 2002.

[BBM+10] Brian Bailey, Felice Balarin, Michael McNamara, Guy Mosenson, Michael Stellfox, and Yosinori Watanabe. *TLM-Driven Design and Verification Methodology*. Lulu Enterprises Inc., 2010.

[BC11] A. Benso and S. Carlos. The art of fault injection. *Journal of Control Engineering and Applied Informatics*, 13(4):9–18, Aug 2011.

[BCB+13] Valeria Bertacco, Debapriya Chatterjee, Nicola Bombieri, Franco Fummi, Sara Vinco, A. M. Kaushik, and Hiren D. Patel. On the use of GP-GPUs for accelerating compute-intensive EDA applications. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 1357–1366, San Jose, CA, USA, 2013. EDA Consortium.

[BD86] George E P Box and Norman R Draper. *Empirical Model-building and Response Surface*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[BFP08] Nicola Bombieri, Franco Fummi, and Graziano Pravadelli. A mutation model for the SystemC TLM2.0 communication interfaces. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 396–401, New York, NY, USA, 2008. ACM.

[BLQ14] Antonio Blin, Youssef Laarouchi, and Philippe Quéré. Fault-injection using virtualization for critical software validation in automotive. In *Embedded Real Time Software and Systems (ERTS^2 2014), Toulouse, France, Feb 2014*, 2014.

[BN98] P. D. Bradley and E. Normand. Single Event Upsets in implantable cardioverter defibrillators. *Nuclear Science, IEEE Transactions on*, 45(6):2929–2940, Dec 1998.

[BP03] A. Benso and P. Prinetto, editors. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, volume 23 of *Frontiers in Electronic Testing*. Springer, Boston, MA, USA, 2003.

[BR10] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, August 2010.

[Bro95] Frederick P. Brooks, Jr. *The Mythical Man-month*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[BSH75] D. Binder, E. C. Smith, and A. B. Holman. Satellite anomalies from galactic cosmic rays. *Nuclear Science, IEEE Transactions on*, 22(6):2675 –2680, dec. 1975.

[Bub13] Chris Bubinas. GM recalls nearly 27,000 vehicles over software problem, April 2013. http://www.klocwork.com/blog/codingstandards/gm-recalls-nearly-27000-vehicles-over-software-problem/ [Online; accessed Dec-2014].

[BVFK05] Raul Barbosa, Jonny Vinter, Peter Folkesson, and Johan Karlsson. Assembly-level pre-injection analysis for improving fault injection efficiency. In *Proceedings of the 5th European Conference on Dependable Computing*, EDCC'05, pages 246–262, Berlin, Heidelberg, 2005. Springer-Verlag.

[BvL14] Dmitry Burlyaev and Rene van Leuken. System fault-tolerance analysis of cots-based satellite on-board computers. *Microelectronics Journal*, 45(10):1335 – 1341, 2014.

[CAMARC+11] Sergio Cuenca-Asensi, Antonio Martínez-Álvarez, Felipe Restrepo-Calle, Francisco R. Palomo, Hipólito Guzmán-Miranda, and Miguel A. Aguirre. Soft core based embedded systems in critical aerospace applications. *Journal of Systems Architecture*, 57(10):886 – 895, 2011.

[CAP+10] Ronald A. Castillo, J. Almena, M. Prieto, D. Guzmán, and S. Sánchez. Validation and Testing of an IP codec for high bandwidth SpaceWire link. In *3rd International SpaceWire Conference*, 2010.

[CC07] K. J. Chang and Y. Y. Chen. System-level fault injection in SystemC design platform. In *Proc. 8th Int. Symposium on Advanced Intelligent Systems*, pages 354–359, 2007.

[CCM+14] E. Carrascosa, J. Coronel, M. Masmano, P. Balbastre, and A. Crespo. XtratuM Hypervisor Redesign for LEON4 Multicore Processor. *SIGBED Rev.*, 11(2):27–31, September 2014.

[CDBA10] Black David C., Jack Donovan, Bunton Bill, and Keist Anna. *SystemC: From the Ground Up, Second Edition*. Springer-Verlag New York, Inc., 2010.

[CDS14] Inc Carbon Design Systems. Carbon model studio, Acton, MA, 2014. http://www.carbondesignsystems.com/ [Online; accessed Dec-2014].

[CG03] Lukai Cai and Daniel Gajski. Transaction level modeling: An overview. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '03, pages 19–24, New York, NY, USA, 2003. ACM.

[CGLH10] J. Manuel Colmenar, Oscar Garnica, Juan Lanchares, and J. Ignacio Hidalgo. Simulating a LAGS processor to consider variable latency on L1 D-Cache. In *Proceedings of the 2010 Summer Computer Simulation Conference*, SCSC '10, pages 56–63, San Diego, CA, USA, 2010. Society for Computer Simulation International.

[Cha09] Robert N. Charette. This car runs on code. *IEEE Spectrum*, Feb 2009.

[Clo10] Electric Cloud. Survey finds 58 % of software bugs result from test infrastructure and process, not design defects, 2010. http://www.electric-cloud.com/news/2010-0602.php [Online; accessed Dec-2014].

[CN13] Domenico Cotroneo and Roberto Natella. Fault injection for software certification. *IEEE Security and Privacy*, 11(4):38–45, 2013.

[Com11] Canadian Nuclear Safety Commission. CNSC Fukushima Task Force Report. Technical Report INFO-0824, Canadian Nuclear Safety Commission, 2011.

[Com14] Australian Semiconductor Technology Company, 2014. http://www.astc-design.com/ [Online; accessed Dec-2014].

[CRM+14] Jerry Carter, Johan Roxendal, Scott McGlashan, Michael Bodell, T.V. Raman, Rahul Akolkar, Marc Helbing, Rafah Hosn, RJ Auburn, Klaus Reifenrath, James Barnett, Noam Rosenthal, Torbjörn Lager, and Daniel Burnett. State chart XML (SCXML): State machine notation for control abstraction. Last call WD, W3C, May 2014. http://www.w3.org/TR/2014/WD-scxml-20140529/.

[dSGCM+09] Antonio da Silva, Alberto Gonzalez-Calero, José Fernán Martínez, Lourdes López, Ana B. García, and Vicente Hernández. Design and implementation of a Java fault injector for Exhaustif. *Dependability of Computer Systems, International Conference on*, 0:77–83, 2009.

[dSMGC+10] Antonio da Silva, José-Fernán Martínez, Alberto González-Calero, Lourdes López, Ana B. García, and Vicente Hernández. XML schema based fault set definition to improve fault injection tools interoperability. *IJCCBS*, 1(1/2/3):220–237, 2010.

[dSML+07] Antonio da Silva, José-Fernán Martínez, Lourdes López, Ana B. García, Luis Redondo, and Vicente Hernández. Exhaustif. A fault injection tool for distributed heterogeneous embedded systems. pages 17:1–17:8, 2007.

[dSPPS] Antonio da Silva, Pablo Parra, Óscar R. Polo, and Sebastián Sánchez. Runtime instrumentation of SystemC/TLM2 interfaces for fault tolerance requirements verification in software cosimulation. *Modelling and Simulation in Engineering*, 2014.

[dSS09a] Antonio da Silva and Sebastián Sánchez. Inyección de fallos en SystemC mediante instrumentación dinámica de código. In *IX Jornadas de computación reconfigurable y aplicaciones*, JCRA2009, 2009.

[dSS09b] Antonio da Silva and Sebastián Sánchez. On the use of dynamic binary instrumentation to perform faults injection in transaction level models. In *Dependability of Computer Systems, 2009. DepCos-RELCOMEX '09. Fourth International Conference on*, pages 237–244, 30 2009-Jul 2 2009.

[dSS09c] Antonio da Silva and Sebastián Sánchez. Transactions sequence tracking by means of dynamic binary instrumentation of TLM models. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on*, pages 723 –728, Aug 2009.

[dSS10] Antonio da Silva and Sebastián Sánchez. LEON3 ViP: A virtual platform with fault injection capabilities. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pages 813–816, sept. 2010.

[dSS11a] Antonio da Silva and Sebastián Sánchez. On the use of debugging with arbitrary record format (DWARF) for symbolic selection of precise and effective time location fault pairs. In *Problems of Dependability and Modelling, book chapter of*, Monographs of System Dependability, pages 51–60. Oficyna Wydawnicza Politechniki Wroclawskiej, Wroclaw, Poland, 2011.

[dSS11b] Antonio da Silva and Sebastián Sánchez. A LEON3 virtual platform with real spacewire interfaces for dependable space software development. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools '11, pages 1–8, ICST, Brussels, Belgium, Belgium, 2011. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[dSSM+10] Antonio da Silva, Sebastián Sánchez, José Fernán Martínez, Ana B. García, Lourdes López, and Vicente Hernández. A grammar based testing framework for TLM2.0 protocol compliance verification. In *Technical Approach to Dependability. Proceedings of RELCOMEX 2010: Fifth International Conference on Dependability of Computer Systems DepCoS*, Monographs of System Dependability, pages 121–132, Wroclaw, Poland, 2010. Oficyna Wydawnicza Politechniki Wroclawskiej.

[dSSPP14] Antonio da Silva, Sebastián Sánchez, Óscar R. Polo, and Pablo Parra. Injecting faults to succeed. Verification of the boot software on-board Solar Orbiter's energetic particle detector. *Acta Astronautica*, 95(0):198 – 209, 2014.

[Dub13] E. Dubrova. *Fault-Tolerant Design*. Springer London, Limited, 2013.

[eCo14] eCos. eCos free open source real-time operating system, 2014. http://ecos.sourceware.org/ [Online; accessed Dec-2014].

[EGVFC+12] L. Entrena, M. García-Valderas, R. Fernández-Cardenal, A Lindoso, M. Portela, and C. López-Ongil. Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection. *Computers, IEEE Transactions on*, 61(3):313–322, March 2012.

[EHK13] Ali Ebnenasir, Reza Hajisheykhi, and Sandeep S. Kulkarni. Facilitating the design of fault tolerance in transaction level SystemC programs. *Theoretical Computer Science*, 496:50–68, 2013.

[EJ09] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, April 2009.

[EMD09] Wolfgang Ecker, Wolfgang Müller, and Rainer Domer. *Hardware dependent Software*. Springer, 2009.

[EP14] Diario El País. Toyota llama a revisión dos millones de Prius por un fallo de "software", 2014. http://economia.elpais.com/economia/2014/02/12/actualidad/1392189800_826724.html [Online; accessed Dec-2014].

[ESA11] ESA. Solar Orbiter. Exploring the Sun-heliosphere. Definition study report. Technical Report I-CA2301, European Space Agency, 2011.

[Esp12] Pablo Parra Espada. *Integración de tecnologías de desarrollo y análisis basadas en componentes bajo un enfoque multi-plataforma*. PhD thesis, Universidad de Alcalá, Departamento de Automática, Alcalá de Henares, Madrid, España, Jul 2012.

[GC06] Jiri Gaisler and E. Catovic. Multi-core processor based on LEON3-FT IP Core (LEON3-FT-MP). In *DASIA 2006 - Data Systems in Aerospace*, 2006.

[GGO03] T. Granlund, B. Granbom, and N. Olsson. Soft error rate increase for new generations of SRAMs. *Nuclear Science, IEEE Transactions on*, 50(6):2065 – 2068, dec. 2003.

[Ghe06] Frank Ghenassia. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[GKS+12] Johannes Grinschgl, Armin Krieg, C. Steger, R. Weiss, Holger Bock, and Josef Haid. Efficient fault emulation using automatic pre-injection memory access analysis. In *SOC Conference (SOCC), 2012 IEEE International*, pages 277–282, Sept 2012.

[GMC+12] D. Ginsberg, A. Mignogna, M. Carloni, F. Menichelli, A. Ferrari, D. Nguyen, and E. Scholte. SystemC based simulation for virtual prototyping of large scale distributed embedded control systems. In *3nd International Workshop on Analysis Tools and Methodologies for Embedded and Realtime Systems (WATERS 2012)*, 2012.

[Gro] The Object Management Group. MDA guide. http://www.omg.org/mda/ [Online; accessed Dec-2014].

[Gro02] Thorsten Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[Gro13] Wilson Research Group. 2012 functional verification study. Technical report, Mentor, 2013.

[Gru13] The COMPLEX reference framework for HW/SW co-design and power management supporting platform-based design-space exploration. *Microprocessors and Microsystems*, 37(8, Part C):966 – 980, 2013.

[GTB+09] Robert Granat, Benyang Tang, Benjamin Bornstein, K. L. Wagstaff, and Michael Turmon. Simulating and Detecting Radiation-Induced Errors for Onboard Machine Learning. In *IEEE International Conference on Space Mission Challenges for Information Technology*, 2009.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[Hen08] Thomas A Henzinger. Two challenges in embedded systems design: predictability and robustness. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366(1881):3727–3736, 2008.

[HOC+12] Sungpack Hong, Tayo Oguntebi, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. A case of system-level hardware/software co-design and co-verification of a commodity multiprocessor system with custom hardware. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '12, pages 513–520, New York, NY, USA, 2012. ACM.

[HSS12] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design. *SIGPLAN Not.*, 47(4):111–122, March 2012.

[HUS14] HUSDSON. Extensible continuos integration server, 2014. http://hudson-ci.org/ [Online; accessed Dec-2014].

[Imp14] Ltd Imperas. Oxfordshire, U.K., 2014. http://www.imperas.com/ [Online; accessed Dec-2014].

[IN11] F. Irom and D. N. Nguyen. SEE and TID response of spansion 512mb NOR flash memory. In *Radiation Effects Data Workshop (REDW), 2011 IEEE*, pages 1–4, Jul 2011.

[Inc14a] CoWare Inc. Coware virtual platform, San Jose, CA, 2014. http://www.coware.com/ [Online; accessed Dec-2014].

[Inc14b] Mentor Graphics Inc. ModelSim/Vista Arquitect, Wilsonville, OR, 2014. http://www.mentor.com/ [Online; accessed Dec-2014].

[Inc14c] Synopsys Inc. Innovator, Mountain View, CA, 2014. http://www.synopsys.com/Tools/SLD/VirtualPlatforms/ [Online; accessed Dec-2014].

[Inc14d] VirtuTech Inc. Simics, San Jose, CA, 2014. http://www.virtutech.com/whatissimics.html [Online; accessed Dec-2014].

[Ini10] Krzysztof Iniewski, editor. *Radiation Effects in Semiconductors*. CRC Press, 2010.

[ITR11] ITRS. SoC design cost model, 2011. http://www.itrs.net [Online; accessed Dec-2014].

[JCS+14] Jiggens, Piers, Chavy-Macdonald, Marc-Andre, Santin, Giovanni, Menicucci, Alessandra, Evans, Hugh, and Hilgers, Alain. The magnitude and effects of extreme solar particle events. *J. Space Weather Space Clim.*, 4:A20, 2014.

[JG09] P. T. A. Jiggens and S. B. Gabriel. Time distributions of solar energetic particle events: Are sepes really random? *Journal of Geophysical Research: Space Physics*, 114(A10):n/a–n/a, 2009.

[JT09] Daniel Jones and Nigel Topham. High speed CPU simulation using LTU dynamic binary translation. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, pages 50–64, Berlin, Heidelberg, 2009. Springer-Verlag.

[Kan11] N. Kanekawa. *Dependability in Electronic Systems: Mitigation of Hardware Failures, Soft Errors, and Electro-Magnetic Disturbances*. SpringerLink : Bücher. Springer New York, 2011.

[KGD+12] M.J. Kay, M.J. Gadlage, A.R. Duncan, D. Ingalls, A. Howard, and T.R. Oldham. Effect of accumulated charge on the total ionizing dose response of a NAND flash memory. *Nuclear Science, IEEE Transactions on*, 59(6):2945–2951, Dec 2012.

[KISU10] N. Kanekawa, E.H. Ibe, T. Suga, and Y. Uematsu. *Dependability in Electronic Systems: Mitigation of Hardware Failures, Soft Errors, and Electro-magnetic Disturbances*. Springer Verlag, 2010.

[LD11] D. Lario and R. B. Decker. Estimation of solar energetic proton mission-integrated fluences and peak intensities for missions traveling close to the sun. *Space Weather*, 9(11):n/a–n/a, 2011.

[Ler07] J.L. Leray. Effects of atmospheric neutrons on devices, at sea level and in avionics embedded systems. *Microelectronics Reliability*, 47(11):1827–1835, 2007. 18th European Symposium on Reliability of Electron Devices, Failure Physics and Analysis.

[LGA+11] Sonia López, Óscar Garnica, David H. Albonesi, Steven Dropsho, Juan Lanchares, and José I. Hidalgo. A phase adaptive cache hierarchy for SMT processors. *Microprocessors and Microsystems*, 35(8):683 – 694, 2011. Design and Verification of Complex Digital Systems.

[LGdVH13] Juan Lanchares, Óscar Garnica, Francisco Fernández de Vega, and José Ignacio Hidalgo. A review of bioinspired computer-aided design tools for hardware design. *Concurrency and Computation: Practice and Experience*, 25(8):1015–1036, 2013.

[LR13] Weiyun Lu and Martin Radetzki. Concurrent and comparative fault simulation in SystemC and its application in robustness evaluation. *Microprocessors and Microsystems*, 37(2):115–128, 2013.

[Lyu96] Michael R. Lyu. *Handbook of software reliability engineering.* McGraw-Hill, 1996.

[MGA+12] Roland Mader, Gerhard Griessnig, Eric Armengaud, Andrea Leitner, Christian Kreiner, Quentin Bourrouilh, Christian Steger, and Reinhold Weiss. A bridge from system to software development for safety-critical automotive embedded systems. *2012 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, 0:75–79, 2012.

[Moo65] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, Apr 1965.

[MRB+11] Juan Antonio Maestro, Pedro Reviriego, Sanghyeon Baeg, Shi-Jie Wen, and Richard Wong. Mitigating the effects of large multiple cell upsets (MCUs) in memories. *ACM Trans. Design Autom. Electr. Syst.*, 16(4):45, 2011.

[MRC+09] Miguel Masmano, Ismael Ripoll, Alfons Crespo, J.J. Metge, and Paul Arberet. Xtratum: An open source hypervisor for TSP embedded systems in aerospace. In *DASIA 2009. Data Systems In Aerospace.*, May. Istanbul 2009.

[MRCM09] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and J.J. Metge. Xtratum: a hypervisor for safety critical embedded systems. In *Eleventh Real-Time Linux Workshop*, Dresden (Germany), September 2009.

[MW79] T.C. May and M.H. Woods. Alpha-particle-induced soft errors in dynamic memories. *Electron Devices, IEEE Transactions on*, 26(1):2 – 9, jan 1979.

[NAS01] NASA. Study on flight software complexity. Technical report, NASA, 2001.

[NBS+02] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Andreas Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *Proceedings of the 39th Annual Design Automation Conference*, DAC '02, pages 22–27, New York, NY, USA, 2002. ACM.

[Nic11] M. Nicolaidis, editor. *Soft Errors in Modern Electronic Systems*. Frontiers in Electronic Testing. Springer, 2011.

[NTC11] Allen Nikora, Kishor Trivedi, and Gianfranco Ciardo. Dependability quantification and assurance of mission-critical software systems. *NASA OSMA Software Assurance Symposium*, 2011.

[OSC14] OSCI. Open SystemC Initiative, SystemC, 2014. http://www.systemc.org [Online; accessed Dec-2014].

[OVP14] Open Virtual Platforms OVP, 2014. http://www.ovpworld.org/ [Online; accessed Dec-2014].

[RBM12] Famantanantsoa Randimbivololona, Abderrahmane Brahmi, and Philippe Le Meur. Airborne software tests on a fully virtual platform. *CoRR*, abs/1204.3410, 2012.

[RBM+14] Famantanantsoa Randimbivololona, Abderrahmane Brahmi, Philippe Le Meur, Thomas Marie, and Romain Beseme. Final integration test of avionic software in full virtual platform. In *Embedded Real Time Software and Systems (ERTS^2 2014), Toulouse, France, Feb 2014*, 2014.

[Rea90] James Reason. *Human Error*. Cambridge [England]; New York: Cambridge University Press, 1990. xv, 302 p., 1990.

[Res13] Gailer Research. Aeroflex Gaisler Research, 2013. http://www.gaisler.com [Online; accessed Dec-2014].

[RMB11] Pedro Reviriego, Juan Antonio Maestro, and Sanghyeon Baeg. Designing ad-hoc scrubbing sequences to improve memory reliability against soft errors. In *Proceedings of the 48th Design Automation Conference, DAC 2011, San Diego, California, USA, June 5-10, 2011*, pages 700–705, 2011.

[RMD03] M. Reshadi, P. Mishra, and N. Dutt. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *Design Automation Conference, 2003. Proceedings*, pages 758–763, Jun 2003.

[RMR11] Oscar Ruano, Juan Antonio Maestro, and Pedro Reviriego. Validation and optimization of TMR protections for circuits in radiation environments. In *14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems, DDECS 2011, Cottbus, Germany, April 13-15, 2011*, pages 399–400, 2011.

[RMRR07] Oscar Ruano, Juan Antonio Maestro, Pilar Reyes, and Pedro Reviriego. A simulation platform for the study of soft errors on signal processing circuits through software fault injection. In *Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on*, pages 3316–3321, Jun 2007.

[RTE14] RTEMS. Rtems real time operating system (RTOS), 2014. http://www.rtems.org/ [Online; accessed Dec-2014].

[Sci05] NASA Science. Radiation resistant computers, November 2005. http://science.nasa.gov/science-news/science-at-nasa/2005/18nov_eaftc [Online; accessed Dec-2014].

[Ser08] Amazon Web Services. Amazon s3 availability event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html [Online; accessed Dec-2014].

[SID14] SIDEMA. Simple SCXML, 2014. http://sscxml.sourceforge.net/ [Online; accessed Dec-2014].

[Sim14] SimpleScalar LLC, 2014. http://http://www.simplescalar.com [Online; accessed Dec-2014].

[SKK+02] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 389–398, Washington, DC, USA, 2002. IEEE Computer Society.

[SKKM13] Martin Straka, Jan Kastil, Zdenek Kotasek, and Lukas Miculka. Fault tolerant system design and SEU injection based testing. *Microprocessors and Microsystems*, 37(2):155 – 173, 2013.

[SL12] V. Sridharan and D. Liberty. A study of DRAM failures in the field. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11, Nov 2012.

[Sor10] J. Sorensen. Space environment and effect analysis section, 2010.

[SPP+13] Sebastián Sánchez, Manuel Prieto, Óscar R. Polo, Pablo Parra, Antonio da Silva, Óscar Gutiérrez, Ronald Castillo, Javier Fernández, and Javier Rodríguez-Pacheco. HW/SW co-design of the instrument control unit for the energetic particle detector on-board solar orbiter. *Advances in Space Research*, 52(6):989 – 1007, 2013.

[SRG12] SRG. *SRG-A3P Development Kit User Manual*. Universidad de Alcalá, Space Research Group, 2012.

[SS12] Marcelo Sousa and Alper Sen. Generation of TLM testbenches using mutation testing. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, pages 323–332, New York, NY, USA, 2012. ACM.

[SWI06] *Software-Implemented Hardware Fault Tolerance*. Springer Verlag, 2006.

[Sys14] Cadence Design Systems. NCSim, San Jose, CA, 2014. http://www.cadence.com [Online; accessed Dec-2014].

[Tea10] Solar Orbiter Team. Experiment Interface Document. Part A, (v2.0), 2010.

[THMC12] Alexandre Trigano, Guillaume Hubert, Jannie Marfaing, and Karine Castellani. Experimental study of neutron-induced soft errors in modern cardiac pacemakers. *Journal of Interventional Cardiac Electrophysiology*, 33(1):19–25, 2012.

[VCM+97] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting how badly "good" software can behave. *IEEE Softw.*, 14(4):73–83, July 1997.

[VM10] P. Reviriego Vasallo and J. A. Maestro. *Radiation Effects in Semiconductors, Soft Errors in Digital Circuits: Overview and Protection Techniques for Digital Filters*. Devices, Circuits, and Systems. Taylor & Francis, 2010.

[VN56] J. Von Neumann. Probabilistic logics and synthesis of reliable organisms from unreliable components. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 43–98. Princeton University Press, 1956.

[WM62] J.T. Wallmark and S.M. Marcus. Minimum size and maximum packing density of nonredundant semiconductor devices. *Proceedings of the IRE*, 50(3):286 –298, march 1962.

[WWZ12] Yang Wang, Lifeng Wang, and Zewei Zheng. Application of virtual prototype technology to simulation test for airborne software system. In *Advances in Electronic Engineering, Communication and Management Vol.2*, volume 140 of *Lecture Notes in Electrical Engineering*, pages 653–658. Springer Berlin Heidelberg, 2012.

[You07] M.T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 23–34, April 2007.

[ZKK11] Claas Ziemke, Toshinori Kuwahara, and Ivan Kossev. An integrated development framework for rapid development of platform-independent and reusable satellite on-board software. *Acta Astronautica*, 69(8):583–594, 2011.

[ZP04] J. F. Ziegler and H. Puchner. *SER History, Trends and Challenges. A guide for Designing with Memory ICs*. Cypress Semiconductor, 2004.

[ZWGC10] Zhaobo Zhang, Zhanglei Wang, Xinli Gu, and K. Chakrabarty. Optimization and selection of diagnosis-oriented fault-insertion points for system test. In *Test Symposium (ATS), 2010 19th IEEE Asian*, pages 429–432, Dec 2010.

Apéndice A

Publicaciones resultado de la tesis

En las siguientes páginas se muestran los principales artículos que se aportan como realización de la tesis.

Artículo 1 [CORE C]
Antonio da Silva and Sebastián Sánchez. On the use of dynamic binary instrumentation to perform faults injection in transaction level models. In Dependability of Computer Systems, 2009. DepCos-RELCOMEX '09. Fourth International Conference on, pages 237-244, 2009.

Artículo 2 [Inspec]
Antonio da Silva and Sebastián Sánchez. LEON3 ViP: A virtual platform with fault injection capabilities. In Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on, pages 813-816, 2010.

Artículo 3 [JCR Q3, SCImago Q1]
Sebastián Sánchez, Manuel Prieto, Óscar R. Polo, Pablo Parra, Antonio da Silva, Óscar Gutiérrez, Ronald Castillo, Javier Fernández and Javier Rodríguez-Pacheco. HW/SW co-design of the instrument control unit for the energetic particle detector on-board solar orbiter. Advances in Space Research, 2013.

Artículo 4 [SCImago Q4]
Antonio da Silva, Pablo Parra, Óscar R. Polo and Sebastián Sánchez. Runtime instrumentation of SystemC/TLM2 interfaces for fault tolerance requirements verification in software cosimulation. Modelling and Simulation in Engineering, 2014.

Artículo 5 [JCR Q2, SCImago Q1]
Antonio da Silva, Sebastián Sánchez, Óscar R. Polo and Pablo Parra. Injecting faults to succeed. Verification of the boot software on-board Solar Orbiter's energetic particle detector. Acta Astronautica, 2014.

A.1. On the use of dynamic binary instrumentation to perform fault injection in transaction level models

Autor

Antonio da Silva y Sebastián Sánchez

Lugar

DepCos-RELCOMEX '09. Fourth International Conference on Dependability of Computer Systems, pages 237-244, 2009

Tipo

Conference paper

Ref

CORE C

Resumen

El modelado en el nivel de transacción (TLM) ha sido comunmente aceptado como estilo de modelado de sistemas. Esta aproximación permite una estimación precisa de aspectos no funcionales y una rápida exploración del espacio de diseño. Además de la simulación funcional para la validación de sistemas HW/SW, existen requisitos adicionales relacionados con la fiabilidad que necesitan de técnicas avanzadas de simulación para analizar el funcionamiento del sistema en presencia de fallos. Los mecanismo tradicionales para inyección de fallos usados en VHDL como "mutantes" o "saboteadores" han sido incorporados a las descripciones de modelos en SystemC. El mayor inconveniente de estas técnicas es la necesidad de disponer del código fuente para realizar las campañas de inyección. En este artículo, se propone el uso de instrumentación dinámica de código (DBI) para realizar la inyección de fallos en modelos SystemC/TLM2. DBI es una técnica para interceptar las llamadas a rutinas software y permite la alteración de los argumentos de la llamada y del valor retornado en tiempo de ejecución. Esta técnica no necesita modificaciones del código fuente ni recompilación del modelo para insertar saboteadores en las transacciones intercambiadas por los componentes.

On the use of Dynamic Binary Instrumentation to perform Faults Injection in Transaction Level Models

A. Da Silva[1] S. Sánchez[2]

[1]*Universidad Politécnica de Madrid*
Crta Valencia, Km 7, 28031, Madrid, Spain
adasilva@diatel.upm.es

[2]*Universidad de Alcalá*
Campus Universitario. Ctra. Madrid-Barcelona, Alcalá de Henares, Spain
ssp@aut.uah.es

Abstract

Transaction Level Modelling (TLM) has been widely accepted as systems modelling framework focused in system components communication. This approach allows efficient accurate estimation and rapid design space exploration. Besides of the functional simulation for validation of a hardware/software designs, there are additional reliability requirements that need advanced simulation techniques to analyze the system behaviour in the presence of faults.

Several traditional VHDL fault injection mechanisms like mutants or saboteurs have been adapted to SystemC model descriptions. The main drawback of these approaches is the necessity of source code modification to carry out the fault injection campaigns.

In this paper, we propose the use of Dynamic Binary Instrumentation (DBI) to perform fault injection in SystemC TLM models. DBI is a technique to intercept software routine calls allowing argument and return value corruption and data structures modification at runtime. This technique needs neither source code modifications nor recompilation of models in order to generate module mutants or in order to insert saboteurs in the signal communication path.

1. Introduction

In today's systems engineering development environments, rapid prototyping and evaluation of fault tolerance capabilities is more important than ever. Thus it is necessary to carry out testing tasks in a very early development stage to ensure that the implemented exception mechanisms work properly. The Transaction Level Modelling (TLM) raises the abstraction level description of a system, focusing in the interchange of data between components through communication channels or sockets. It can be used to describe systems at different levels of abstraction, from untimed functional to cycle accurate models. The primary goal of TLM was to allow early software development and to join the hardware and software design flow.

C and C++ are languages widely known and have been a popular starting point for describing hardware designs and systems. Those systems are quick to write and can give an executable version of the specification, which allows a very fast simulation. Plus, versions of standard are broadly available, so it is possible to easily reuse legacy and publicly available code. For system-level design, these languages allow to describe hardware and software components in a single framework. Furthermore, development tools are just C++ compilers, debuggers, and development environments that hardware/software designers are already familiar with.

1.1. TLM in SystemC

SystemC [1] provides an industry-standard means of modelling and verifying hardware and systems using standard software C++ compilers such as Microsoft Visual C++ or GNU GCC. SystemC was released to the public in Sept. 1999 by the Open SystemC Initiative (OSCI). OSCI TLM 2.0 [2] has been released in Jun 2008 and its goal is to ease and enable interoperability between high-level SystemC components. It defines modelling styles, several interfaces and a generic payload for transactions. There are two modelling styles:

- The Loosely Timed (LT) style: is targeted for system and platform models, where timing and data are only loosely connected. This modelling style does not specify a level of abstraction but specifies the functionality supported.
- The Approximated Time (AT) style: It is used for systems where the dependency of timing and data is very strong. Any timing dependencies between components can be explicitly modelled.

1.2. Related work

Several fault injection techniques for fault tolerance coverage testing in critical systems have been considered. In old systems to perform fault injection, physical or hardware implemented fault injection [3] consisting on signal modifications at pin level were used in very late phases of the design cycle. With the use of hardware description languages like VHDL, others fault injection techniques have been used, especially those based on the use of *"saboteurs"* and *"mutants"* in VHDL models [4]. *"Saboteurs"* are additional modules inserted into the signal path between two components and *"Mutants"* are new versions of a module that replace the original. Normally these models describe the systems at Register Transfer Level (RTL) and the runtime simulation is very slow, being impossible to perform hardware-software co-simulation.

The works presented in [5] introduces fault injection methods for SystemC-based systems descriptions. One of the main drawbacks presented in many fault injection scenarios is the time overhead introduced; in order to improve the performance of executable models in the presence of faults, some strategies are shown to accelerate the SystemC simulation by parallel computing. [6] presents system-level fault injection in SystemC. The proposed framework of fault injection consists on untimed functional TLM modelling with FIFO channels.

Mutation analysis and mutation testing have gained consensus during the last years as being one of the most important techniques for software testing. Since SystemC provides an executable model for systems descriptions, mutations modules can be used to test the exception handling mechanisms. Bombieri [7] proposes a mutation model for perturbing transaction level modeling (TLM) SystemC descriptions. In particular, the main constructs provided by the SystemC TLM 2.0 library have been analyzed, and a set of mutants is proposed to perturb the primitives related to the TLM communication interfaces.

Intercepting routine calls to introduce some kind of corruptions in incoming parameters, as well as for return values, is a very common approach in Software Implemented Fault Injection tools. The tool described in [8] uses Dynamic Binary Instrumentation to intercept Operating Systems Calls under Windows in order to test applications robustness.

The approach presented in this work addresses the problem of inserting "*saboteurs*" in the transaction path with a minimal time overhead and without modifications in the TLM source code description. This makes the technique useful for the validation of third party Intellectual Property (IP) cores which can be distributed as binaries.

The remainder of the paper is organized as follows: an approach of DBI techniques for C++ binaries is introduced and briefly analyzed in section 2. Section 3 describes the experimental setup used to evaluate the proposed approach. Section 4 concludes this paper with the obtained conclusions.

2. Dynamic Binary Instrumentation and C++ binaries

Dynamic binary instrumentation (DBI) is a technique to intercept system calls to analyze the runtime behaviour of software. This approach is based on modifying the target API call with a ***jump*** instruction to the user defined wrapper function. This wrapper function will have the functionalities the user wants to add. This approach replaces the first few instructions of the original target function and stores them in a function called "trampoline function," which is called by the wrapper function after its processing is finished. This mechanism works fine for functional programming languages like C but for object oriented languages like C++ other approach must be used.

2.1. C++ objects memory layout

Each time a class that contains virtual functions is created or it is derived from a class that contains virtual methods [9], the compiler creates a unique virtual method address table (VTABLE) for that class, see figure 1. In that table it places the addresses of all the functions that have been declared virtual in this class or in the base class. This means that the method address invoked is obtained at runtime. It is possible to modify the VTABLE for a specific object instance and insert the address of a function wrapper to perform runtime modification of the transaction parameters and monitor the behaviour of the system on the presence of faults.

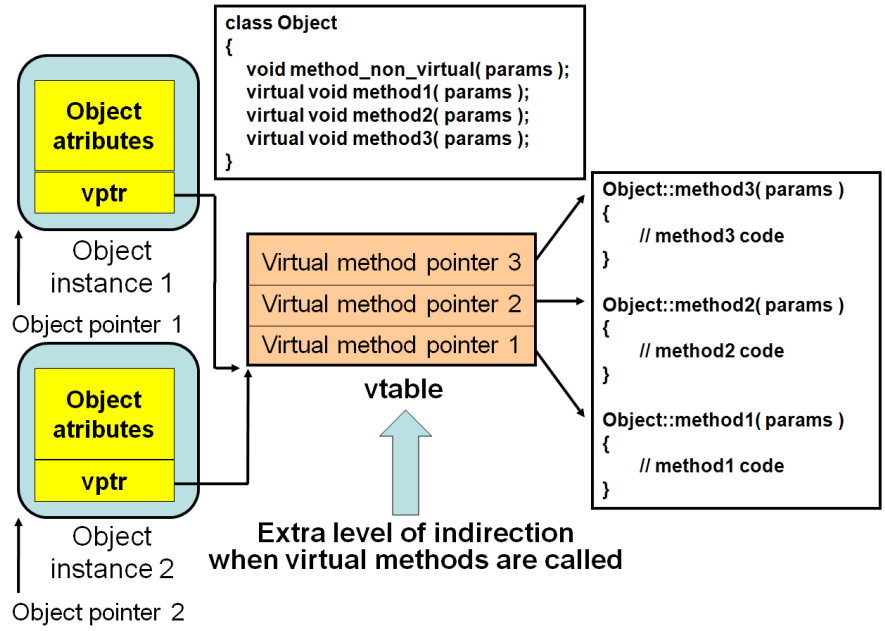


Figure 1. C++ Virtual Methods call Mechanism.

2.2. Simple target socket hierarchy

Although the hooking method proposed works only for virtual methods this is a very common situation. In objects hierarchy is often to find base classes which present only an interface for its derived ones. Actually no instance of a base class is created, only a description of an interface is given. This is done in C++ by making the base class abstract, which means that at least one method is declared pure virtual. When an abstract class is inherited, all pure virtual method must be implemented, or the inherited class becomes abstract as well. Creating a pure virtual method allows to describe an interface without being forced to provide an implementation. The derived class must provide its own specific version of the virtual method.

```
class tlm_blocking_transport_if  :  public virtual sc_core::sc_interface
{
public:
  virtual void b_transport( TRANS& trans,  sc_core::sc_time& t)  =  0;

};
```

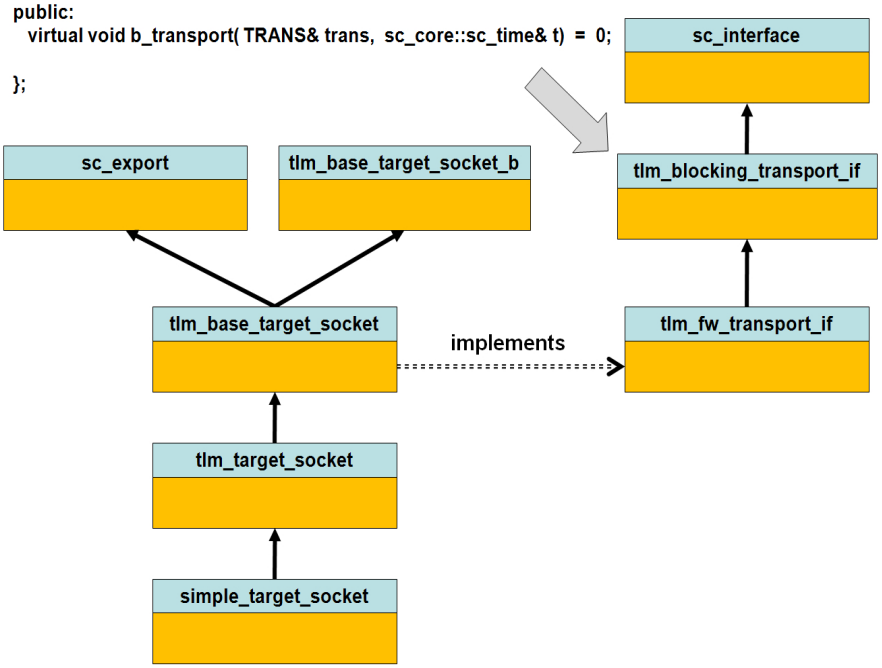


Figure 2. TLM2 Hierarchy for blocking transport.

The basic hierarchy of the TLM2.0 *"simple_target_socket"* is shown in figure 2. One of the transport interfaces supported is *"tlm_blocking_transport_if"*. This interface defines only one pure virtual method called *"b_transport"*. Every instance of a *"simple_target_socket"* must provide its own implementation of the *"b_transport"* method. Due to the fact that this method is virtual, every call is made through the VTABLE of the object instance and so it is possible to modify it to insert some kind of function wrapper without modifying the original source code.

3. Experimental setup

A simple testing environment was built using SystemC, see figure 3, in order to measure the processing time overhead introduced and thus validate the usefulness of the hooking technique proposed in this paper. One transaction initiator binds to a transaction target and uses the blocking transport interface *"b_transport"* to carry out transactions. In order to intercept the transaction between both modules two approaches were used:

- Use of an Interposition Module: in this case the initiator is bound to a module inserted in the transaction path. Is important to keep in mind that doing this, a source code modification is required in order to bind the initiator to the interceptor and doing the same between the interceptor and the target.
- VTABLE hooking: in this case the virtual table of the initiator module is modified to call a function wrapper. This function will finally call the original "*b_transport"* implementation of the target. The initiator and target modules were bound one to another and no source code modification is necessary to intercept the transaction. Even more, to obtain a pointer to the initiator object instance, the standard SystemC API can be used; it returns an object pointer given its name.

Both test cases work in a pass-through mode and just forward the incoming transaction request to the target. The aim of the test is to measure the time overhead introduced. No transaction parameters corruptions are done because the time spent would be the same for both cases. For each initiator socket transaction the time spent is measured using the time-stamp counter present in Intel processors [10]. The time-stamp counter keeps an accurate count of every cycle that occurs on the processor.

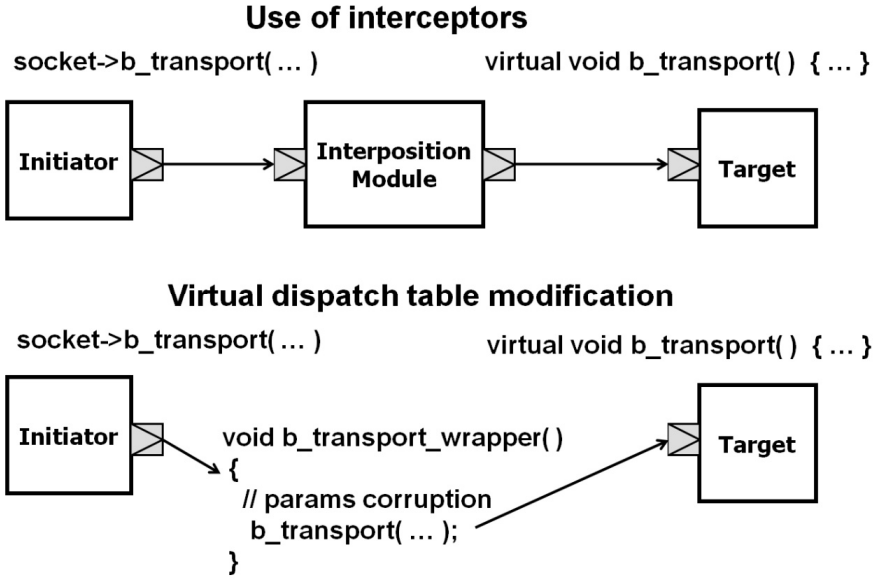


Figure 3. Interception Mechanisms.

3.1. Wrappers Insertion Code Snippets

Next are the coding steps necessary to insert a function wrapper in the calling path of a TLM2.0 non blocking interface using Microsoft's Visual C++ compiler. In order to keep the example code simple, runtime errors are not handled in the description.

```
1.  typedef tlm::tlm_fw_transport_if<tlm::tlm_base_protocol_types> fw_if_type;
2.  typedef tlm::tlm_blocking_transport_if<tlm::tlm_base_protocol_types::tlm_payload_type> block_transp_if_type;
3.  typedef tlm::tlm_base_protocol_types::tlm_payload_type transaction_type;
4.  typedef void (__stdcall *BTransportPtr)(transaction_type&, sc_core::sc_time&);
5.  BTransportPtr ptr_b_transport_org;

6.  __declspec( naked ) void b_transport_wrapper( transaction_type& trans, sc_core::sc_time& delay )
 {
    // Preprocessing wrapper code, be carefull with C++ calling conventions specially "this" pointer
    ptr_b_transport_org( trans, delay );        // Original b_transport call
    // Postprocessing wrapper code
 }
```

Figure 4. Basic types and Wrapper Function Definition.

Figure 4 shows the basic types definition. Particular attention must be paid to the definition of the ***ptr_b_transport_org*** function pointer, see lines 4 and 5. This pointer will hold the address of the original transport method and it is used inside ***b_transport_wrapper*** implementation to maintain the original call path, see function wrapper code starting from line 6. Preprocessing of incoming parameters and postprocessing of returned values could be done inside the wrapper code. Figure 5 describes how to change the virtual dispatch table of a TLM2.0 initiator socket. Code in line 3 obtains a pointer to the SystemC object given its name. From lines 4 to 6, a pointer to the concrete ***tlm_blocking_transport_if*** implementation is obtained. As is explained in section 2, the virtual table address is stored at the beginning of the object instance. Thus, using object pointer is possible to obtain virtual table beggining, see line 7. Lines 8 and 9 make a copy of the original virtual table and assign it to the object instance. Finally original method address is saved in ***ptr_b_transport_org*** in line 10 and line 11 changes the virtual table entry by inserting the function wrapper address.

```
1.  ULONG *p_vtable;
2.  ULONG NewVTable[2]; // There is only one virtual method in tlm_blocking_transport_if
                        // Last position is filled with NULL
3.  sc_object* p_object = sc_find_object("initiator.socket_0"); // Find an object given its hierarchical name

4.  sc_port<fw_if_type>   *p_port_socket = dynamic_cast<sc_port<fw_if_type> *>(p_object);
5.  sc_interface          *p_interface = p_port_socket->get_interface();
6.  block_trans_if_type   *p_block_trans_if_type = dynamic_cast<block_trans_if_type *>(p_interface);
7.  p_vtable = *((ULONG *)p_block_trans_if_type);

8.  for( int i =0; i<2; i++ ) {  NewVTable[i] = p_vtable[i];  }          // Copy original table
9.  *((ULONG *)p_block_trans_if_type) = (ULONG)NewVTable;      // Assign the new VTABLE to the object
10. ptr_b_transport_org = NewVTable[0];
11. NewVTable[0] = (ULONG) b_transport_wrapper;  // Change the first entry inserting the wrapper address
```

Figure 5. Wrapper Insertion Procedure.

3.2. Performance issues

In order to obtain representative statistics, it is necessary to carry out not only one transaction but several. There are some issues that could affect the cycle count so the time measurement. The same section of code can often produce very different results due to data and instruction cache. It is important to repeat several times the measurement to find the average real-time execution time of a piece of code, in this case a method call.

As is shown in figure 4 about 1000 calls to *"b_transport"* target method was done using three different configurations, see figure 3:

- Normal transaction path between one initiator and a target.
- Use of virtual table hooking to interpose a function wrapper in transaction path.
- Use of an interposition module between the initiator ant the target.

Al tests were carried out using a 1,66 Ghz generic laptop with 1 Gbyte RAM Memory under Microsoft Windows XP Professional Service Pack 2 and the results values are summarized in table 1.

Table 1. Cycles Measurement.

	Average cycles	Cycles overhead
Normal Path	4600	
Virtual table hooking	4700	2%
Interposition module	6215	35%

As has been said before no corruption of transaction parameters was introduced in these tests and only the time cost of inserting a parameters "*saboteur*" was considered. The results shown in figure 4 reveals that virtual table hooking introduces a minimum time overhead in the normal transaction path. This is very important to maintain the simulation speed of a system within acceptable values. Even more this can be accomplished without source code modification. Another important point to consider is the time spent inserting or removing the wrapper that can be considered inappreciable. This makes easier the faults activation/deactivation process.

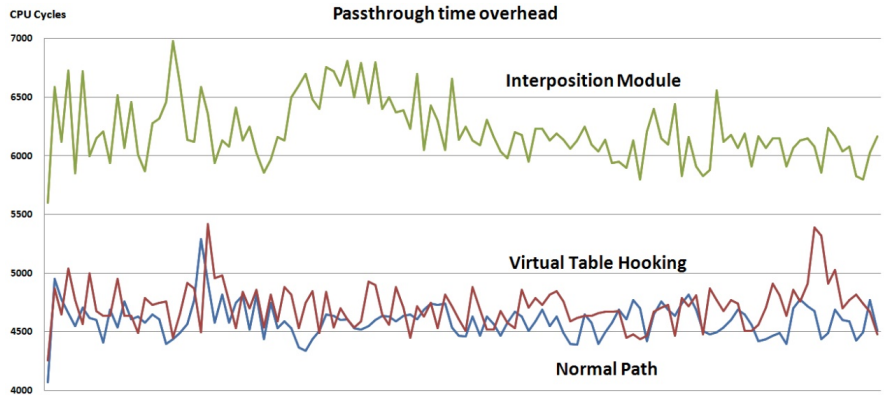


Figure 6. Measured Cycle Overhead.

4. Conclusions and future work

Transaction Level Modelling has become the main methodology to handle the complexity challenges in nowadays system level design. TLM raise the abstraction level enabling faster design space exploration at early design stages and hence reducing the "time to market" time.

The capacity of injecting faults if essential for the verification of fault tolerance mechanisms which are foreseen in the construction of critical systems, as well as to predict the consequences of the bad systems behaviour due to errors not detected in functional tests.

The early results here presented are encouraging, and shows that is possible to insert transaction "saboteurs" in an easy way with a minimal time overhead and with a great improvement over previous approaches like interception modules. In addition, the technique here adopted is applicable in third party software component validation, without having access to component source code.

It is necessary to confirm the results obtained in these experiments, continuing on testing the techniques by inserting "*saboteurs*" and injecting faults into more complex models.

An API for transaction parameters corruption in TLM2.0 descriptions is now under development. It will help to automate the verification process because no source code modification is need. All fault injection campaign could be carried out over a functional well working executable model.

5. References

[1] Open SystemC Initiative, http://www.systemc.org. SystemC.

[2] Open SystemC Initiative, http://www.systemc.org/downloads/standards/tlm20/.

[3] Benso, A., Prinetto, P., eds., "Fault Injection Techniques and Tools for VLSI reliability evaluation", Kluwer Academic Publishing, 2003.

[4] Baraza, J.C., Gracia, J., Gil, D., Gil, P.J., "Improvement of Fault Injection Techniques Based on VHDL Code Modification", Proceedings of the High-Level Design Validation and Test Workshop, 2005, pp. 693-706.

[5] Misera, S., Vierhaus, H. T., Sieber, A., "Fault Injection Techniques and their Accelerated Simulation in SystemC", Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools, 2007, pp. 587-595

[6] Chang, K.J., Chen, Y.Y., "System-Level Fault Injection in SystemC Design Platform", Proceedings of the 8TH Symposium on Advanced Intelligent Systems, 2007

[7] Bombieri, N., Fummi, F., Pravadelli, G. "A Mutation Model for the SystemC TLM 2.0 Communication Interfaces", Proceedings of the conference on Design, automation and test in Europe, 2008, pp. 396-401.

[8] Da Silva, A., Martínez J.F., Lopez, L., García A.B., Hernández, V., "XML Schema Based Faultset Definition to Improve Fault Injection Tools Interoperability", Proceedings of International Conference on Dependability of Computer Systems. DepCos-RELCOMEX 2008, pp. 39-46.

[9] Thinking in C++, Volume 1, 2nd Edition, January 13, 2000 Bruce Eckel, President, MindView, Inc.

[10] Using the RDTSC Instruction for Performance Monitoring, http://developer.intel.com/

A.2. LEON3 ViP: A virtual platform with fault injection capabilities

Autor

Antonio da Silva y Sebastián Sánchez

Lugar

13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD), ISBN: 978-1-4244-7839, 2010.

Tipo

Conference paper

Ref

Inspec

Resumen

En el desarrollo de software espacial, hay requisitos de robustez y tolerancia a fallos que necesitan de técnicas y herramientas de simulación avanzadas para analizar el comportamiento del sistema en presencia de fallos. Además es deseable que estas herramientas posean la capacidad de comunicar el software embebido bajo desarrollo con otros sistemas reales usando interfaces de comunicación estándar. En este artículo se presenta el diseño de una plataforma virtual para LEON3, un procesador SPARC de 32 bits usado por la agencia espacial europea. Esta plataforma se ha descrito en el nivel de transacción usando SystemC/TLM2. La plataforma permite comunicaciones SpaceWire reales con otros equipos. Cada componente TLM2 proporciona una interfaz "transport_dbg" que permite la inspección y modificación interna del componente. De esta manera es posible llevar a cabo campañas de inyección de fallos alternado el contenido de los registros del procesador así como el contenido de la memoria.

LEON3 ViP : a Virtual Platform with Fault Injection capabilities

Antonio da Silva
DIATEL-EUIT Telecomunicación
Universidad Politécnica de Madrid
Madrid, Spain
adasilva@diatel.upm.es

Sebastián Sánchez
Departamento de Automática
Universidad de Alcalá
Alcalá de Henares, Spain
ssp@aut.uah.es

***Abstract*—In addition to functional simulation for validation of hardware/software designs, there are additional robustness requirements that need advanced simulation techniques and tools to analyze the system behavior in the presence of faults. In this paper, we present the design of a fault injection framework for LEON3, a 32bit SPARC CPU based system used by the European Space Agency, described at Transaction Level using SystemC. First of all an extension of a previous XML formalization of basic binary faults, like memory and CPU registers corruption, is done in order to support TLM2.0 transaction's parameters corruptions. Next a novel Dynamic Binary Instrumentation (DBI) technique for C++ binaries is used to insert fault injection wrappers in SystemC transaction path. For binary faults in model components the use of TLM2.0 "transport_dbg" is proposed. This way each component with fault injection capabilities exposes a standard interface to allow internal component inspection and modification.**

Keywords-component; LEON3, Fault Injection, XML Faultset, Binary Instrumentation, Debug Transport Interface.

I. INTRODUCTION

In today's systems design, rapid prototyping and evaluation of expected behavior is more important than ever. Thus it is necessary to carry out testing tasks in a very early development stage to ensure that the implemented exception mechanisms work properly and helps to evaluate the risks, revealing how the system behaves in the presence of faults. These fault tolerance requirements ask for integrated, easy to use, full simulation environments, where Instruction Set Simulators (ISS) allow software to be developed and tested with a high accuracy in a very early hardware development stage. This is essential to evaluate fault detection and recovery mechanisms implemented in the software design.

Virtual platforms are software models of complete systems that provide software engineers with development environments long time before real hardware is available. Virtual platforms enable concurrent development of SoC hardware and software, significantly shortening their integration time. For embedded software development, virtual platforms provide faster edit-compile-debug cycles through more controllability, observability and determinism in the carried out experiments.

System-level modeling languages are used to start the design process from a more abstract level and applying a top-down design methodology. The use of SystemC [1] provides an industry-standard mechanism of modeling and verifying hardware and systems using standard C++ compilers. The Transaction Level Modeling (TLM)[2] raises the abstraction level description of a system, focusing in the interchange of data between components through abstract communication channels or sockets. Due to their advantages, TLMs have been traditionally used for design space exploration, early architectural performance estimations and to allow an earlier software development start, joining the hardware and software design flow.

The remainder of the paper is organized as follows: relevant related works are briefly depicted in section 2, section 3 describes the proposed framework; first of all an approach for fault taxonomy definition based on XML Schema is introduced, next the techniques and interfaces for fault injection are described. Section 4 describes the experimental setup used to evaluate the proposed approach. Section 5 contains the conclusions.

II. RELATED WORK

The work presented in [3] introduces fault injection methods for SystemC-based systems descriptions. One of the main drawbacks presented in many fault injection scenarios is the time overhead introduced; in order to improve the performance of executable models in the presence of faults, some strategies are used to accelerate the SystemC simulation by parallel computing.

SystemC provides an executable model for system description, so mutations modules can be used to test the exception handling mechanisms. Bombieri [4] proposes a mutation model for perturbing transaction level modeling (TLM) SystemC descriptions. In particular, the main constructs provided by the SystemC TLM 2.0 library have been analyzed, and a set of mutants is proposed to perturb the primitives related to the TLM communication interfaces.

SimSoC [5] presents the design of an ARM based system. The framework integrates ISSs as SystemC modules with other platform components by means of TLM interfaces. It uses dynamic translation of the binary code to speed up execution.

RESP framework [6] is so far the most complete framework for virtual platform building. The use of Python language provides a powerful mechanism to analyze the given SystemC description and extract the set of public object attributes where faults could be injected. The main drawback of this approach is that those public attributes not necessarily reflect a functional view of the component, but a programmers view. Even more, good programming practices suggest those attributes to be private and provide a public API in order to access them.

The framework presented in this work is a specific LEON3 virtual platform with fault injection capabilities. First of all, a XML fault taxonomy of the fault model is proposed. This approach makes the fault description independent of application GUI and fault injection technique. Second, for components internal state corruption, we proposed the use of TLM2.0 "transport_dbg" call in order to access components internal state. Each component must expose their internal functional attributes, allowing framework inspection and modification. For transaction level corruption we propose the use of DBI in order to insert runtime binary wrappers in the transaction path.

III. FRAMEWORK DESCRIPTION

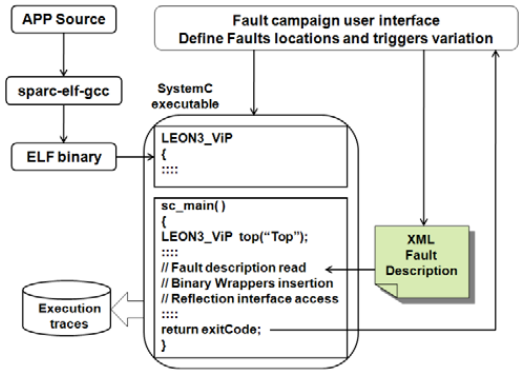


Figure 1. Proposed Framework.

Figure 1 shows the basic structure of the proposed framework. It is build around the following components:

A. Basic Fault taxonomy and XML formalization

TABLE I. TLM2.0 TRANSACTIONS' PAYLOAD FAULTS

Where	*What*	*When*	*Last*
b_transport call -Command -Address -Data -Response status	Mask value BitFlip value Ad-hoc value	Always Time Trigger Signal Trigger Transact. number Logical Combo	Always Time Trigger Signal Trigger Transact. number Logical Combo
nb_transport_fw call -Command -Address -Data -Response status -Phase -TLM2_Sync	Mask value BitFlip value Ad-hoc value	Always Time Trigger Signal Trigger Transact. number Logical Combo	Always Time Trigger Signal Trigger Transact. number Logical Combo
nb_transport_bw call -Command -Address -Data -Response status -Phase -TLM2_Sync	Mask value BitFlip value Ad-hoc value	Always Time Trigger Signal Trigger Transact. number Logical Combo	Always Time Trigger Signal Trigger Transact. number Logical Combo

The work [7] describes a fault's taxonomy for binary and resources of host based systems. This approach can be used to describe memory and ISS registers corruptions by applying different kind of corruptions patterns like bitflips or bitmasks. Finally, triggers are used to define when the fault must be injected and how long it takes. It describes all the conditions that should be met for a fault to become active or not. This allows defining not only permanent and transient faults but also intermittent faults.

In order to complete this approach, Table I describes a fault's taxonomy for TLM2.0 transaction interface. Three corruption patterns have been defined: masking, bitfliping and assigning an ad-hoc value. These corruptions patterns can be applied to the TLM2.0 parameters of blocking/non blocking calls: "b_transport", "nb_transport_fw" and "nb_transport_bw".

B. LEON3 Transaction Level Model

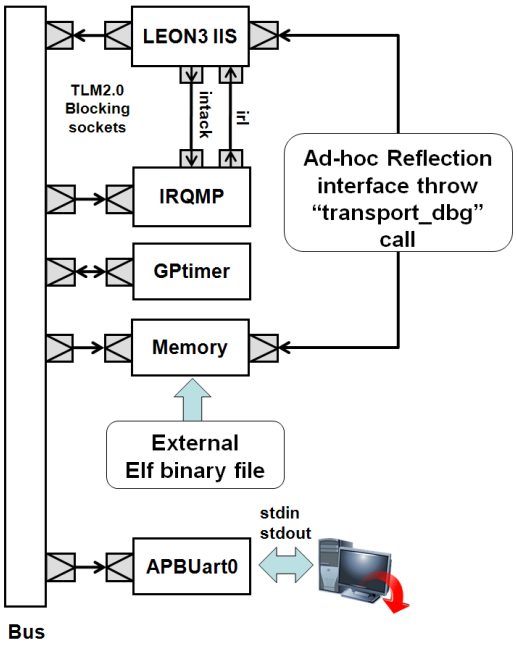


Figure 2. LEON3 basic Model.

Figure 2 show the transaction level model of the LEON3 system. The minimal building blocks are:

- LEON3 ISS: Sparc V8 untimed ISS.
- GPTimer: Basic timer capabilities.
- IRQMP: Interrupt controller.
- Memory: 2Mbyte PROM beginning at address 0x00000000 and 4Mbyte RAM starting at 0x40000000. The memory contents are read from an external ordinary ELF file generated by the compiler toolchain.
- APBUart0: The interface is provided for serial communications. It is used by software to perform standard input/output.

C. Reflection Interface definition

Every LEON3 ViP module with fault injection capabilities exports an ad-hoc reflection interface accessible by means of TLM2.0 “transport_dbg” call. Throw this interface the components expose a functional view of the internal architecture of the component regardless how the component has been coded. Carrying out “read/write” operations is possible to get/set the actual state of the component. For memory modules the functional view reflects the memory map. For LEON3 module, reflection interface allows accessing the architectural elements of the ISS like program counters, registers set and so on.

D. Runtime Transaction Wrappers Insertion

For interface fault injection, transaction wrappers in TLM2.0 sockets interface can be inserted at runtime without top module modifications, depending on the specific fault set to be injected and described by the XML file. After the elaboration phase, all the elements are instantiated, the initiators sockets have been bound to targets sockets. Next and before the simulation starts, runtime wrappers are inserted using a novel virtual table hooking technique described in [8][9]. This technique can be applied after the elaboration phase is done and needs neither source code modifications nor recompilation of top level models descriptions. Once the transaction path is intercepted by means of a wrapper, this can be used in several kinds of scenarios like: transaction tracking or snooping, fault injection by corrupting transaction parameters, transactions’ assertions insertion, protocol verification, etc.

IV. Experimental setup

In order to test the overall proposed framework, the following scenario has been implemented. As target application the system runs a SHA-1 hash calculator. The SHA hash functions are a set of cryptographic hash functions designed by the National Security Agency (NSA)[10] and published by the NIST as a U.S. Federal Information Processing Standard. SHA algorithms set are intended to be employed in a wide variety of security applications, and are also commonly used to check the integrity of files. They are supposed to replace the well known MD5 hash. This kind of application is chosen because it has a high processing workload and it is easy to perform the calculation of a given input data and therefore knowing the expected results. The original source code is compiled using the “C” toolchain provided by Gaisler Research [11].

A. Fault Model

For each memory section present in elf binary a complete fault injection campaign is carried out, sweeping memory address from the beginning to the end of the section. For injection time the same approach is used. From zero to end simulation time several injection points are uniform distributed along the execution time. This way the temporal and spatial influence of the faults can be explicitly stated. For each address and injection time the corresponding XML fault description is written, the virtual platform is launched and the exit code obtained.

B. System behaviour

For each fault injected in the model, the behavior of the application under test is checked. Each exit code describes one of following behaviors:

- No influence: Application runs and ends properly. The calculated hash is as expected.
- Bad hash: Application seemingly runs and ends properly but the result hash is wrong. This is the worst situation since in a real scenario there is no hint that there was a failure. Analyzing the results can help in protecting the most sensitive parts of the application.
- Memory read/write error. The faults lead the system to out of memory space access due to a bad address calculation. This kind of failures is detected and can be corrected. In the worst case a system reset is issued.
- Application Hang. The injected fault leads code to an endless loop. This situation can also be detected by watch-dog like mechanisms.
- Integer Unit (IU). This kind of failures happens only when fault affects TEXT section. In this case the original opcode is modified and the new one is an invalid opcode.

C. Detailed Results

More than 268.000 experiments were carried out in the fault injection campaign. Faults were injected across all typical system memory sections: BSS, DATA, HEAP and TEXT. The BSS section contains all global and static variables that are initialized to zero by default. On the other hand, DATA section contains global and static variables that are not initialized to zero. The last data section is the HEAP area which begins at the end of the BSS segment and grows to larger addresses available on the system. Finally the TEXT section contains the application code. For all the faults injected only about 17.000 have some kind of effect on the program behavior. According to the results obtained the most sensitive section is TEXT.

Figure 3. DATA section fault injection results.

There are only bad hash failures for faults injected in DATA section, see figure 3. This is an expected result since DATA section contains global variables and probably is used for temporary results during the final hash calculation. As was previously highlighted this is the worst behavior, so this section must be especially protected.

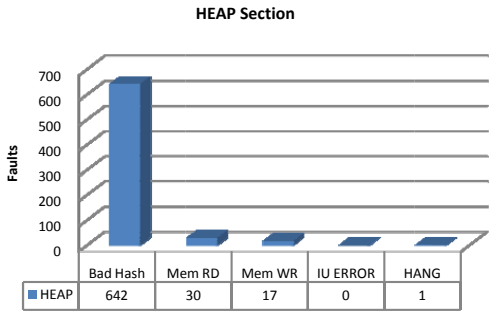


Figure 4. HEAP section fault injection results.

HEAP section, figure 4, experiments several kind of failures. Most of them are bad hash like DATA section. However there are 47 out of memory access and 1 application hang. In these cases the injected faults have effect on memory pointers and control loop variables.

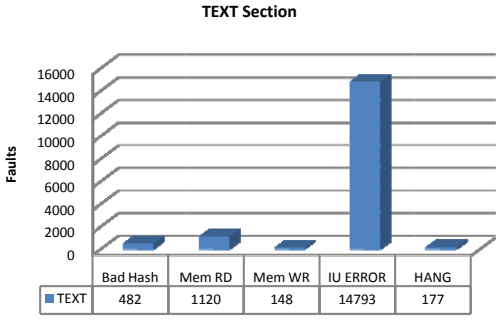


Figure 5. TEXT section fault injection results.

TEXT section results are shown in figure 5. In this case most of the failures are IU Errors. This means that opcode corruptions usually generate invalid opcodes. However there are a wide number of situations where the new opcode is a valid one and the side effects are unpredictable. Above 1.250 out of memory access and 177 application hang are the failures detected by hardware. Again the worst situations are the 482 faults that provoke no hardware errors but bad hash calculations.

V. CONCLUSIONS

The capacity of injecting faults is essential for the verification of fault tolerance mechanisms that are foreseen in the construction of critical systems, as well as to predict the consequences of the bad systems behavior due to errors not detected in functional tests.

This is a first modeling approach of basic fault sets in mixed TLM models descriptions and components internal state modification. It can be easily extended to other faults and corruptions scenarios. XML fault sets are easy to read and compare not only for machines but also for humans. It facilitates communication and collaboration and improves experiment reproducibility.

To corrupt the internal state of the model's components, the definition of well known interfaces is needed. In order to perform controlled corruptions in architectural components' elements it is necessary to access functional views of those components. The runtime wrapper insertion technique used in the framework presented here shows that it is possible to insert transaction "saboteurs" in an easy way with a minimal time overhead and with a great improvement over previous approaches like interposition modules. In addition, the technique here adopted is applicable to third party software component validation, without having access to component source code. In addition, since insertion/removal of such "saboteurs" is time-consuming and error-prone, automating this task also ensures that the testing process does not compromise the correctness of the final system.

Analysis of experiments' results indicates that it is feasible to indentify strategic locations where faults have more catastrophic consequences and hence improving the software fault tolerance capabilities.

REFERENCES

[1] Open SystemC Initiative, http://www.systemc.org. SystemC.

[2] Open SystemC Initiative, http://www.systemc.org/downloads/standards/tlm20/.

[3] Misera, S., Vierhaus, H. T., Sieber, A., "Fault Injection Techniques and their Accelerated Simulation in SystemC", Proceedings of the 10th Euromicro International Conference on Digital System Design, DSD 2007, pp. 587-595

[4] Bombieri, N., Fummi, F., Pravadelli, G. "A Mutation Model for the SystemC TLM 2.0 Communication Interfaces", Proceedings of the conference on Design, automation and test in Europe, 2008, pp. 396-401.

[5] Helmester, C., Joloboff, V., "SimSoC: A SystemC TLM integrated ISS for full system simulation", Proceedings of International Asia Pacific Conference on Computer Architecture and Systems, 2008.

[6] Beltrame, G., Fossati, L., Sciuto, D., "ReSP: A Nonintrusive Transaction-Level Reflective MPSoC Simulation Platform for Design Space Exploration", in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2009, Volume: 28 Issue:12, pp. 1857 – 1869.

[7] Da Silva, A., Martínez J.F., Gonzalez-Calero, A., Lopez, L., García A.B., Hernández, V., "XML Schema Based Faultset Definition to Improve Fault Injection Tools Interoperability", Int. J. Critical Computer-Based Systems, Vol. 1, Nos. 1/2/3, pp.220-237 (2010).

[8] Da Silva, A., Sánchez, S., "On the use of Dynamic Binary Instrumentation to perform Faults Injection in Transaction Level Models", Proceedings of International Conference on Dependability of Computer Systems. DepCos-RELCOMEX 2009, pp. 237-244.

[9] Da Silva, A., Sánchez, S., "Transactions Sequence Tracking by means of Dynamic Binary Instrumentation of TLM Models", Proceedings of the 12th Euromicro International Conference on Digital Systems Design. DSD 2009, pp. 723-728.

[10] http://tools.ietf.org/html/rfc1374 (2010)

[11] http://www.gailer.com (2010)

A.3. HW/SW Codesign of the Instrument Control Unit for the Energetic Particle Detector onboard Solar Orbiter

Autor

Sebastián Sánchez, Manuel Prieto, Óscar R. Polo, Pablo Parra, Antonio da Silva, Óscar Gutiérrez, Ronald Castillo, Javier Fernández y Javier Rodríguez-Pacheco

Lugar

Advances in Space Research, ISSN: 0273-1177, 2013, Volumen: 52, Página: 989-1007

Tipo

Journal paper

Ref

SCImago 2013: Category Aerospace Engineering Q1
JCR 2013: Q3

Resumen

La misión Solar Orbiter de la ESA está concebida para realizar un estudio detallado de nuestro Sol y la parte interior de la heliosfera para entender mejor el comportamiento de nuestro Sol. La misión proporcionará las claves para descubrir cómo el Sol crea el viento solar y cómo éste afecta a los entornos de todos los planetas. La nave está equipada con un extenso conjunto de instrumentos. El Detector de Partículas Energéticas (EPD) es uno de los instrumentos in situ a bordo de Solar Orbiter. EPD se compone de cinco sensores diferentes y todos ellos comparten la Unidad de Control del Instrumento o ICU, que es la única interfaz con la nave espacial. En este trabajo se hace hincapié en cómo el enfoque de codiseño hardware/software puede conducir a una disminución de la complejidad del desarrollo software y pone de manifiesto la versatilidad del conjunto de herramientas que apoyan el proceso de desarrollo. Siguiendo un enfoque de ingeniería basado en modelos, estas herramientas son capaces de generar el código de alto nivel de la aplicación, así como de facilitar su control de configuración y su despliegue en las plataformas de hardware utilizados en las diferentes etapas del proceso de desarrollo. Por otra parte, el uso de la plataforma virtual "Leon2ViP", con capacidades de inyección de fallos, permite una verificación temprana del software en ausencia del hardware y también una cosimulación hardware/software. Las soluciones adoptadas reducen el tiempo de desarrollo sin poner en peligro toda la fiabilidad del proceso, que es esencial para el éxito de EPD.

HW/SW Co-design of the Instrument Control Unit for the Energetic Particle Detector on-board Solar Orbiter[☆]

Sebastián Sánchez*, Manuel Prieto, Óscar R. Polo, Pablo Parra, Antonio da Silva, Óscar Gutiérrez, Ronald Castillo, Javier Fernández, Javier Rodríguez-Pacheco

Space Research Group, Universidad de Alcalá, Alcalá de Henares, Madrid, Spain

Abstract

ESA's medium-class Solar Orbiter mission is conceived to perform a close-up study of our Sun and its inner heliosphere to better understand the behaviour of our star. The mission will provide the clues to discover how the Sun creates and controls the solar wind and thereby affects the environments of all the planets. The spacecraft is equipped with a comprehensive suite of instruments. The Energetic Particle Detector (EPD) is one of the in-situ instruments on-board Solar Orbiter. EPD is composed of five different sensors, all of them sharing the Instrument Control Unit or ICU that is the sole interface with the spacecraft. This paper emphasises on how the hardware/-software co-design approach can lead to a decrease in software complexity and highlights the versatility of the toolset that supports the development process. Following a model-driven engineering approach, these tools are capable of generating the high-level code of the software application, as well as of facilitating its configuration control and its deployment on the hardware platforms used in the different stages of the development process. Moreover, the use of the Leon2ViP virtual platform, with fault injection capabilities, allows an early software-before-hardware verification and validation and also a hardware-software co-simulation. The adopted solutions reduce development time without compromising the whole process reliability that is essential to

[☆]This work has been supported by the MINECO under the grants AYA2011-29727-C02-01 and AYA2011-29727-C02-02.

*Corresponding author

Email address: sebastian.sanchez@uah.es (Sebastián Sánchez)

the EPD success.

Keywords:
Energetic Particle Detector, Space instrumentation, Data processing, Flight software, Software verification and validation, Component-based software development, Fault injection, Virtual platform

---

1. Introduction

Solar Orbiter medium-class mission is the first ESA mission to be launched within the Cosmic Vision program (ESA, 2012). It aims to study the Sun from a 0.28 AU position (as close as the orbit of Mercury) and out of the ecliptic plane. This mission scenario will enable the spacecraft to both view the Sun from close-in and to view its polar regions. Solar Orbiter will include a set of telescopes to image the Sun and a complementary set of instruments to sample the outflowing solar wind. Some of these instruments will also sample the electromagnetic fields and charged particles emitted from the solar surface and ejected out into interplanetary space. Combining these measurements with the observations of activity on the solar surface will help Solar Orbiter scientists to discover how processes on the Sun create and control the interplanetary environment and ultimately affect the Earth and other planetary systems.

The Energetic Particle Detector (EPD) is an in-situ instrument suite that consists of five sensors measuring electrons, protons, and ions from helium to iron, and operating at partly overlapping energy ranges from 2 keV up to 200 MeV/n. The EPD sensors are located externally in different places of the spacecraft. The five mentioned sensors are:

- SupraThermal Electrons, Ions, & Neutrals (STEIN)
- Suprathermal Ion Spectrograph (SIS)
- Electron Proton Telescope (EPT)
- Low Energy Telescope (LET)
- High Energy Telescope (HET)

The EPD sensors share the Instrument Control Unit (ICU) that is composed of the Common Data Processing Unit and the Low Voltage Power

Supply (CDPU/LVPS). The ICU is the sole power and data interface of EPD to the spacecraft.

STEIN, located at the boom, consists of a single unit having two view cones in opposite directions. SIS consists of two sensor heads with roughly opposite (160°) view directions sharing a common electronics box. EPT and HET sensors are grouped together in a single box where they share a common electronics. The combined unit is called EPT-HET and it has multiple view cones. There are two identical EPT-HET units. LET has multiple view cones and consists of two separate identical units. The overall energy coverage achieved with the EPD sensors is 0.002 MeV to 20 MeV for electrons, 0.003 MeV to 100 MeV for protons, 0.008 MeV/n to 200 MeV/n for heavy ions (species-dependent). This energy and species coverage well satisfies, and for a large part exceeds, the requirements defined for EPD in the Solar Orbiter Payload Definition Document and in the report of the Joint Science and Technology Definition Team (JSTDT) for the Solar Orbiter/Sentinels mission (JSTDT, 2008).

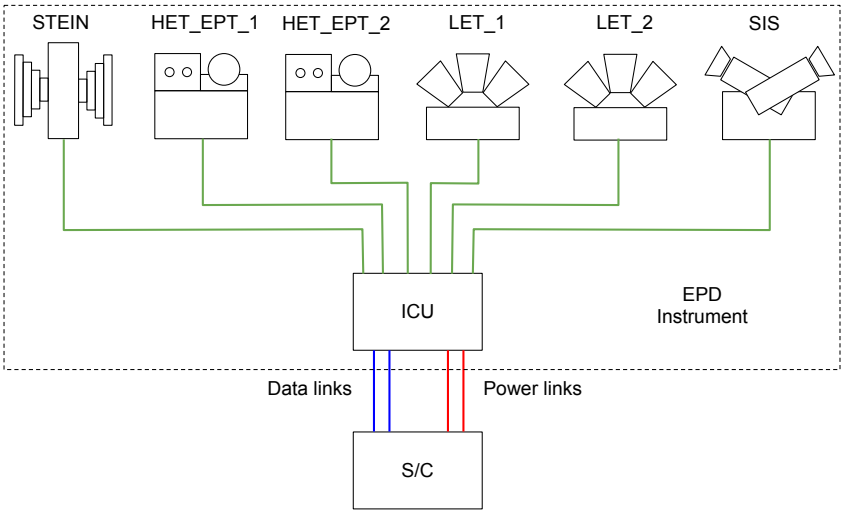


Figure 1: EPD instrument interfaces.

EPD Instrument Control Unit

As it can be seen in Figure 1, all EPD sensors are connected to the ICU, which provides data communication/processing and power to all the sensors. Therefore, the elements included in the EPD's ICU can be grouped into two main functions, implemented in two units, the CDPU and the LVPS. The former is responsible for data processing and communications and its design is based on the utilization of a FPGA with a LEON2 processor (Gaisler Research, 2005) synthesized in it. The later is responsible for providing the power supply to the CDPU and the sensors. The ICU provides a single-point digital interface to the spacecraft and to all EPD sensors. It also shares information with other Solar Orbiter instruments to allow synchronized high data rate burst-mode operations following the on-board identification of predefined triggering events in the EPD data. The ICU boards are packaged in a single box to reduce mass, to simplify harnessing and interfaces, and to reduce duplication in common services. The ICU electronics box is mounted inside the spacecraft body. Electronic parts for the ICU were selected according to criteria for reliability, size, functionality, radiation hardness, and low-power. Parts were specified to be tested to achieve a minimum total dose radiation tolerance requirement of 100 krads. Each sensor has its own front-end electronics. Sensors communicate with the ICU through Universal Asynchronous Receiver Transmitter/Low-Voltage Differential Signalling (UART/LVDS) interfaces operating at 115200 bauds. Sensors provide synchronized data to the ICU on 1-second period except SIS that sends data on 3-second period.

The ICU is responsible of providing the following functions:

- To provide the power supply to the sensors. The ICU is able to drive the +28v of the spacecraft to the sensors through Latch Current Limiters (LCL) allowing to switch on/off the sensors and monitoring current and voltage. It also provides over-current and under-voltage protection.

- To perform scientific and housekeeping data acquisition from the sensors. When the sensors are in nominal mode they provide scientific and housekeeping data packets with a cadence of one second. Only SIS operates at the lower cadence of three seconds. In order to synchronize this operation, the ICU provides a 1Hz (1/3Hz for SIS) hardware clock to the sensors.

- To receive the EPD telecommands and route them, if necessary, to the sensors and to packetise the telemetry according to the ECSS Packet Utilisation Standard (PUS) (ECSS Secretariat, 2003a) standard. A special PUS service is used to perform the synchronization among Solar Orbiter instruments. Instruments will send a regular packet to the On-Board Computer (OBC), containing data for sharing with the other instruments. This is sent at a maximum rate of 8Hz, via PUS Service 20. The OBC will then distribute the received information among the different instruments.

- To process and monitor the information provided by the sensors. Scientific data shall be compressed, stored and formatted to be sent to the spacecraft. Housekeeping data shall be monitored, stored, and also formatted before sending it as housekeeping telemetry.

- To manage the *burst mode* of the instrument. The burst mode is activated when a special event is raised internally within EPD, or externally by other Solar Orbiter instruments. These events are associated to specific physical phenomena such as solar flares, which explosively release magnetic energy, driving shocks and accelerating particles. When one of such events is raised, the resolution of the data shall be higher than in nominal mode of operation. When the burst mode is triggered, the time resolution is increased during 15 minutes. In nominal operation the time resolution is 10 seconds for all the sensors but SIS, which has a 30-second resolution. In burst mode, the time resolution is increased by a factor of 10. Thus the time resolution is 1 second for all sensors but for SIS, which is 3 seconds.

- To implement the EPD Fault Detection Isolation and Recovery (FDIR) mechanism. The faults are always handled at the lowest level but, if this level is not capable of recovering the system from the fault, it forwards it to an upper level. For instance, if one sensor is not able to recover by itself, the fault is forwarded to the ICU and if the ICU is not able to recover from it, it is forwarded to the spacecraft. Those faults that cannot be solved on-board, are sent to ground.

The ICU consists of four electronics boards, two boards (nominal & redundant) for the CDPU and two boards (nominal & redundant) for the LVPS. The nominal and redundant parts of both the CDPU and LVPS are identical

and work in a cold redundant configuration. In this way, the reliability of the system is increased although more mass and volume are needed. In Figure 2, the redundancy within the ICU is shown. Note that nominal and redundant +28V power lines from the spacecraft are connected to nominal and redundant LVPS units, while nominal and redundant SpaceWire data lines are connected to nominal and redundant CDPU units. The connection with one generic sensor is also shown. Blue lines from the nominal and redundant CDPUs to the sensors contain UART/LVDS data links and hardware timing signals. Red lines from nominal and redundant LVPSs provide +28V to the sensors through LCLs. Blue lines and red lines that connect the ICU with the sensors share the same harness. The harness connecting the ICU with all the sensors has the same signals: power, data transmission, data reception and timing, together with the redundant counterparts.

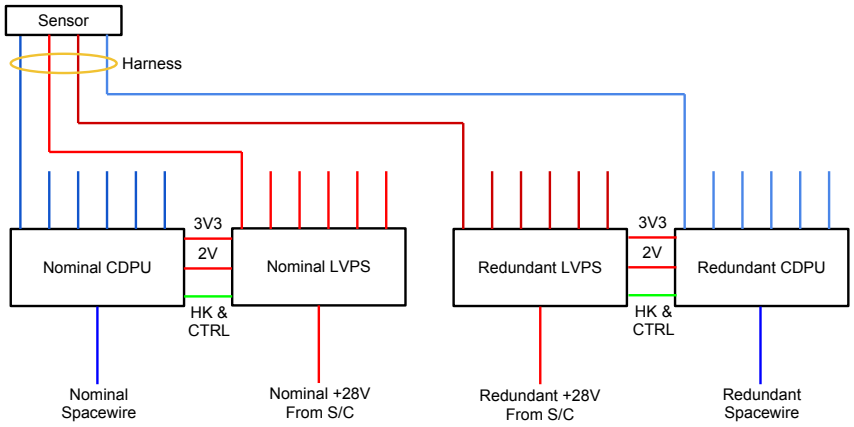


Figure 2: Redundancy within the ICU.

Considering the ICU's functionality, the development of the CDPU has been carried out following a hardware/software co-design approach. In this way, the whole set of system requirements has been considered holistically, incorporating both the hardware and the software parts, in order to analyse which partitioning and design decisions can simplify the complete solution without compromising the reliability of the final product.

After the partitioning, the software development has followed a model-driven engineering process supported by a set of specific tools developed by

the Space Research Group (SRG) of the University of Alcala. These tools allow, on one hand, the graphical modelling of the interfaces and behaviour of the high-level software entities, as well as the connections established among them, and the generation of their implementation code in C++. Furthermore, the tools incorporate different models to be used for configuring and deploying the software on top of the deployment platforms that will be used during the system life cycle.

One of these platforms is based in an environment called Leon2ViP (da Silva and Sánchez, 2010), also developed internally by the SRG. This environment consists of a simulator of the LEON2 processor that allows the injection of hardware failures as well as the co-simulation of the software with functional models of the IP-Cores incorporated in the system during the co-design stage. The use of this simulated environment allows an early software-before-hardware verification and validation, facilitating the fault tolerance tests and enabling the checking of the system recovery from possible hardware failures.

The rest of the paper is organized in the following sections: Section 2 provides a detailed discussion of the hardware and software co-design of the ICU and the design of the CDPU defined after the partitioning. The next section includes the main characteristics of the software that controls the ICU together with a description of its development and verification and validation process. The two final sections describe the related work and the most relevant conclusions of this paper.

2. ICU HW/SW co-design

A general schema of the ICU HW/SW co-design approach is shown in Figure 3. The set of software entities that are part of the ICU application software communicate among them through a software bus and indirectly with the IP Cores using specific control modules. The software modules are, in turn, implemented as tasks or interrupt handlers. A real-time operating system (RTOS) provides task management support and the basic synchronisation mechanisms, such as mutex and signals, needed to implement the software bus. IP Cores communicate among them through the Advanced Microcontroller Bus Architecture or AMBA bus (ARM, 1999).

Following the co-design paradigm, the overall system functionality is distributed among software entities and hardware IP Cores. The partitioning decision is the result of a trade-off process between the availability of third-party IP Cores, the logic resources availability of the FPGA and the possible

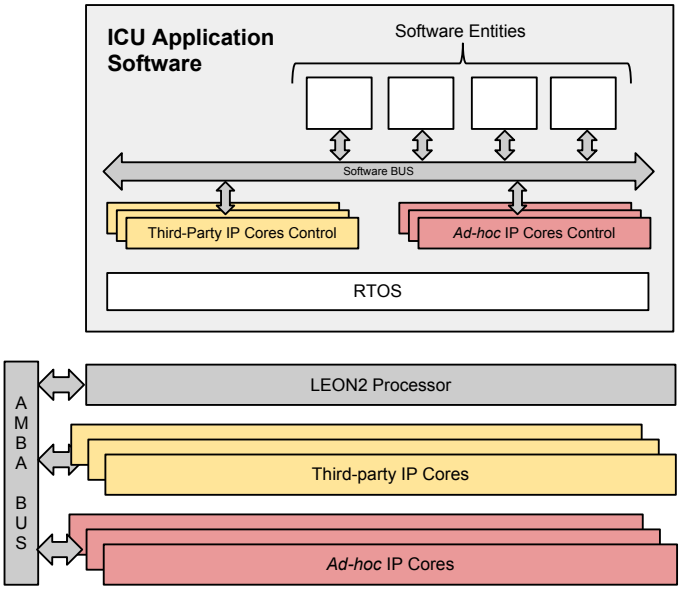


Figure 3: Schema of the ICU HW/SW co-design approach.

impact in terms of complexity and performance of the implementation by means of software entities.

In the case of the ICU, the key requirements that have mainly influenced the partitioning are the following:

- The ICU uses specific communication channels to receive the telemetry sent by the sensors, as well as the telecommands sent by the spacecraft. The LVDS communication channels with the sensors operate at 115200 bauds, while the spacecraft can send bursts of telecommands through the SpaceWire link at 10Mbps. The specifications of the communication protocol determine, moreover, that the telemetry packets sent by the sensors might be of variable length, while the size of the telecommands received from the spacecraft is at most 248 bytes plus the SpaceWire standard header.

- The ICU has to distribute among the sensors the one pulse per second (1PPS) signal synchronized with the reception of the SpaceWire time-

code sent by the spacecraft. The 1PPS signal, however, must also be distributed autonomously if a timecode has not been received after one second and 200 microseconds.

According to the first requirement, the maximum frequency with which each sensor sends the telemetry characters is approximately 10kHz, while the telecommands received from the spacecraft corresponding to the different telecommands arrive at most every 250 microseconds. From this information, it can be easily deduced that a solution based on interrupt handlers triggered by the reception of one piece of data would have had as a result an excessive workload that could have affected the performance of the entire application software. Therefore, a hardware mechanism is needed to autonomously store the received data. While it could have been implemented as a managed buffer integrated in the very IP core, that solution would have required a considerable amount of space in the FPGA to store all the data at the required frequencies. Another solution, which is the one that has been finally opted for in this project, is to implement an *ad-hoc* design of the IP Core that uses a DMA to store the received data in memory. The main details of this solution are described in the next subsection.

Regarding the second requirement, the use of a specific timer is needed due to the fact that the time interval for switching from an autonomous distribution of the 1PPS signal and the one triggered by the arrival of the timecode through the SpaceWire interface, must not exceed 200 microseconds. The adopted solution has been to modify the IP Core that manages the timer so that it also distributes the signals and performs the switching in case the timecode was not received.

Both decisions have thus required the addition to the CDPU of IP Cores that have been designed *ad-hoc* and have prevented an intensive use of interruptions and simplified the software design and, ultimately, its validation.

2.1. CDPU

The CDPU provides the basic intelligence to EPD, controlling the data flow between the instrument and the spacecraft. It receives commands from the spacecraft and manages the sensors. It has a LEON2 soft-processor implemented in an Actel RTAX-2000SL FPGA, running at 20.0 MHz, which provides 17 MIPS performance. The FPGA not only includes the soft-processor, but also includes the following IP Cores: UART/LVDS interface with the sensors, 1Hz hardware clock generator that synchronizes the sensors with the

SpaceWire time-codes received from the spacecraft, the interface with the LVPS, SpaceWire interface with the spacecraft, the glue logic, PROM, EEPROM, SDRAM and Error Detection and Correction (EDAC) controllers, Direct Memory Access (DMA) controllers, and a watchdog. Furthermore, during the testing phase, a debug UART interface also instantiated inside the FPGA, allows the connection of external diagnostic terminals.

The CDPU also contains PROM, EEPROM and SDRAM memories, one SpaceWire link and six identical serial sensor links (see Figure 4). Serial sensor links connect the CDPU to the EPD sensor suite (STEIN, HET-EPT, LET and SIS), providing them with commanding, timing, and telemetry services. The CDPU PROM (a 32Kx8 memory based on the Aeroflex UT28F256 part) contains boot code that can load flight code from either the EEPROM or from the spacecraft interface via telecommand (in the event of an EEPROM failure or corruption). The EEPROM (a 256Kx32 memory based on the 3D-Plus 3DEE8M32VS8094 MCM part) contains two copies of the flight code. The SDRAM (a 64Mx40 memory that uses a 3DSD2G40VS5238 module with EDAC manufactured by 3D-Plus) is used for storing code, variables, and buffers. Most of the CDPU SDRAM is used during burst mode to save high time resolution interval snapshots from STEIN, HET-EPT, LET and SIS, which are then transferred at a slower rate via telemetry.

Figure 4 shows an overview of the main subsystems comprising the CDPU.

Processor

The ICU central processing unit is a 32-bit LEON2 processor. LEON2 is a 32-bit RISC high performance synthesizable processor core conforming to the IEEE-1754 (SPARC V8) standard. The core is highly configurable, and particularly suitable for system-on-chip (SOC) designs. The system is organized around two on-chip buses based on the AMBA bus. The first is the AMBA High-performance Bus (AHB) and the second is the AMBA Advanced Peripheral Bus (APB). The system uses the AHB bus to connect the LEON2 processor to the memory controller, and to other high-bandwidth IP Cores such as SpaceWire. Serial and JTAG debug IP Cores are also connected to this bus. By default the processor is the only master on the bus, while at least two slaves are provided: a memory controller and an APB bridge. The memory controller provides access to those memories used in the system, i.e. PROM, EEPROM and SDRAM. The APB Bridge is connected to the AHB bus as a slave and acts as the master on the APB bus. It is used to access on-chip registers in the peripheral functions, such as UART, interrupt

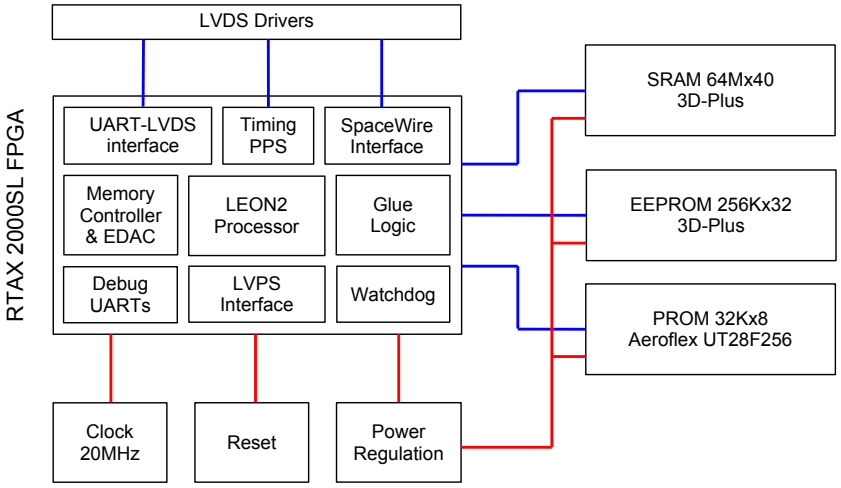


Figure 4: ICU Common Data Processing Unit block diagram.

controllers, timers and a general purpose I/O ports. New modules can be easily added by using the on-chip AMBA AHB/APB buses.

LEON2 provides the necessary support to control and monitor the sensors and the data flow, to collect housekeeping data from the sensors, and to perform processing. A watchdog reset pulse is generated if the LEON2 does not write to the Watchdog Reset Clear register for a period of a given number of seconds.

Memory

The memory is organized in three banks, PROM, EEPROM and SDRAM. PROM memory stores boot code that is in charge of performing sanity checks to the EEPROM and SDRAM memories. In case an error is detected in EEPROM memory, the CDPU's boot code is able to patch this memory via telecommands received through the SpaceWire link. EEPROM memory contains the application software (two copies) and the sensor's calibration tables. Finally, SDRAM memory is used to store scientific and housekeeping data, tasks' stacks and heap. SDRAM is protected against single event upsets thanks to the use of an EDAC combined with memory scrubbing algorithms.

Spacecraft interface

The ICU-spacecraft interface is based on a SpaceWire IP Core implemented in the FPGA and operating at 10 Mbps. This link is used as an interface to transmit housekeeping and scientific data, and also as a command interface to receive telecommands from the spacecraft. Transfer of scientific and housekeeping data shall take place according to the CCSDS recommendations defined in (CCSDS, 2000) and (ECSS Secretariat, 2003a). EPD command messages are also embedded in CCSDS telecommand packets. The spacecraft forwards CCSDS telecommand packets to the EPD when they contain any of the EPD identifiers or Application Identifiers (APID) (one for each unit). The CDPU strips off the CCSDS packet headers and analyses the destination identifier. If the identifier is associated to the CDPU, the telecommand is processed directly by the CDPU. If the identifier is associated to one of the sensors, the CDPU decodes and forwards the command to the specific sensor. Command responses from the sensors are sent to the CDPU, which in turn transmits them to the spacecraft in the form of CCSDS telemetry packets. Command responses generated by the CDPU itself are also CCSDS formatted and transmitted to the spacecraft.

Sensors data interface

EPD sensors' commands and telemetry are transmitted via UART/LVDS interfaces at 115200 bauds. These UARTs are implemented as IP Cores in the FPGA. A cold redundant configuration will be used for all transmission/reception signals between the CDPU and the sensors. Data communication between the sensors and the CDPU is started every second by the CDPU, which acts as a master. Special care has been taken in the process of handling the data received from the sensors. Operating every channel at 115200 bauds, in the worst case, approximately every 90 μs a character is received and an interrupt is raised per serial line. Due to the fact that the six units can generate and transmit data at the same time, the characters and thus the interrupts can arrive at a rate of one character every 15 μs. This high interrupt rate can provoke a big overhead on the processor. For this reason, a DMA controller has been included in the FPGA. Thanks to this, the received characters are moved from the UART to the memory without processor intervention. This approach allows the reduction of the processor overhead and simplifies the software.

Sensors timing synchronization

CDPU provides a 1Hz hardware clock (one pulse per second or 1PPS) to the sensors. The 1PPS signal is generated by an IP Core and synchronized with the spacecraft through the time-codes provided by the SpaceWire link in accordance with (ECSS Secretariat, 2003b). The CDPU drives the 1PPS signal directly to the sensors via a cabled hardware signal. LVDS will be used to send the 1PPS clock signal, in a cold redundant configuration, to the sensors. This approach avoids the software latency and the jitter of the processor's interrupt response. Note that all the signals (transmission/reception and 1PPS clock), both nominal and redundant, are differential signals.

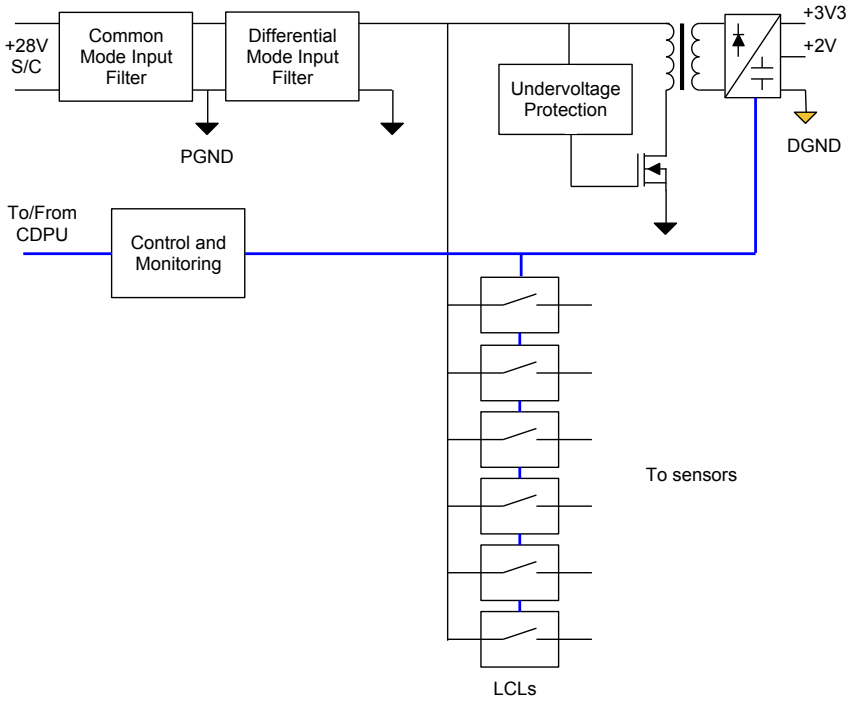


Figure 5: ICU Low Voltage Power Supply block diagram.

2.2. LVPS board

The LVPS board is responsible for filtering, monitoring and switching the spacecraft's primary power to the sensors (see Figure 5). It also provides the power supply to both CDPUs, nominal and redundant. It must be pointed out that the LVPS is responsible of distributing the primary power to the sensors. Input filters are included in order to comply with the ElectroMagnetic Compatibility (EMC) requirements, filtering the emissions produced from the LVPS CDPU DC-DC converter.

The LVPS includes under-voltage and short-circuit protection mechanisms for the sensors. In the former case, the LCLs responsible for switching on/off the sensors are only enabled if the input power bus voltage is above certain level and are automatically switched off if this power bus voltage drops below it. In the later case, if a short-circuit is produced in the primary power of any sensor, the corresponding LCL limits its current during a few milliseconds. After that, the LCL is switched off.

3. ICU software design and development process

The ICU software (ICUSW) is responsible of EPD's command and data handling. It manages the system's start-up, the TM/TC interfaces with the spacecraft, the interfaces with the sensors, the mode management, the error handling and the data processing.

The ICUSW consists of two well-differentiated elements: an application software (ICUAppSw), whose development is strongly tied to the co-design process detailed in the previous section; and a boot software (ICUBoot), which executes basic configuration and hardware checking tasks before deploying the application software into RAM. Their development has been planned using advanced software engineering techniques that allow the reduction of costs and mitigate the risks of delays in the project milestones. Both elements are thoroughly described in the next paragraphs, as well as the model-driven process followed and the tools used in their development. The process is based on a platform-aware approach and is supported by a specific framework developed in-house. Furthermore, the component-based graphical modelling and code generation techniques used to develop the highest-level software entities of the ICUAppSw is also described. Finally, an insight is given into how the Leon2ViP environment has been used to verify the behaviour of the ICUBoot in the face of hardware failures, as well as to perform

tests based on the co-simulation of the software with the functional model of one of the IP cores that has been developed *ad-hoc* after the partitioning.

3.1. ICU Boot Software

ICUBoot is responsible for managing the EPD's basic configuration and hardware checking. It is also in charge of deploying a valid ICUAppSw image, stored in EEPROM, to the SDRAM memory, and later on giving the control to it if the sanity checks are successfully passed. Each ICUAppSw image is organized in segments whose integrity is supervised by Cyclic Redundancy Check (CRC), so that any possible damage of the EEPROM would only require the repair of the affected segments. In order to increase the system's reliability, the ICUBoot handles two EEPROM ICUAppSw images that can be repaired by both the ICUBoot and by the ICUAppSw itself while running in SDRAM. If none of the images is valid, ICUBoot keeps the control of the EPD until the spacecraft restores the integrity of the images and an ICU reboot is commanded. Figure 6 shows an scheme of this process.

This reliability mechanism can be exhaustively tested thanks to the fault injection platform Leon2ViP, based on SystemC, which is described in Subsection 3.2. ICUBoot software has been integrated into a framework, called MICOBS (Parra et al., 2011), that enables a platform-aware model driven engineering approach. The integration allows to manage its execution on the fault injection platform, during reliability qualification test, and on the different ICU prototypes, during the development milestones until the final software is transferred to the ICU's flight model.

3.1.1. ICUBoot platform-aware model-driven engineering

MICOBS is a development framework that has been fully designed and developed by the SRG. One of the key elements of the MICOBS framework is that it includes the hardware deployment platform as an ubiquitous design dimension. Following a platform-aware approach, all the different software elements that can be defined within the framework must specify the platform or platforms on top of which they can be executed. Furthermore, all the generative processes that take place within MICOBS during the different development stages, are mainly platform-driven, i.e., they take into account the platform when generating the different products, whether they are final or intermediate.

MICOBS defines two integration levels: packaging and composition. The packaging level provides support for the deployment and configuration of

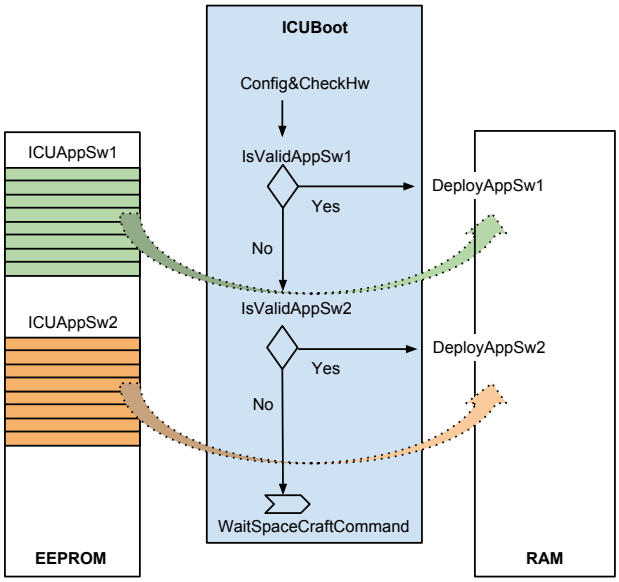


Figure 6: ICUBoot management of the ICU Application Software deployment.

embedded software projects during the different development stages. Within this level, software applications are divided into reusable modules called *software packages* that are compiled and linked together to form the different software applications. These packages can provide and require services to and from other software packages by means of an abstraction called *software interface*. Apart from stating the interfaces they provide and/or require and a possible set of configuration parameters, packages must declare, following the MICOBS' platform-aware approach, the hardware deployment platforms on top of which they can be executed. The meta-data of the software packages and the interfaces are stored together in plain files along with the rest of the code, version-tagged and configured in Subversion repositories. Listing 1 shows an example of the description model of a software package, `ccsds_protocol_slib`, with the code that implements the CCSDS layer functionalities. Apart from the name and version of the package, the model

includes the list of languages and construction mechanisms it supports; the interface it provides, namely `ccsds_protocol_iface`; and the platforms on top of which it can be executed. In this example, the package can be run on RTEMS operating system using the regular RTEMS API and also on a Linux machine using the POSIX API. In both cases, the package does not depend on the actual hardware of the platform. Finally, the package can define a set of parameters whose values will be set at deployment time and that will be use to appropriately configure it.

Listing 1 `ccsds_layer` software package.

```
swpackage ccsds_protocol {
 version := v1;
 languages := C(C99);
 construction tools := GNUMake(3.81);
 provided interfaces {
   provides ccsds_protocol_iface(v1) {};
 };
 supported platforms {
   supported platform RTEMSAPI_4_6_RTEMS_4_6_6_any_any_any {
     osapi := RTEMSAPI(4.6);
     os := RTEMS(4.6.6);
     architecture := any;
     microprocessor := any;
     board := any;
   };
   supported platform POSIX_v13_Linux_2_6_any_any_any {
     osapi := POSIX(v13);
     os := Linux(2.6);
     architecture := any;
     microprocessor := any;
     board := any;
   };
 };
 configuration parameters {
   integer PUS_TM_PACKET_MAX_SIZE := 4106;
   integer PUS_TC_PACKET_MAX_SIZE := 242;
 };
};
```

Following this approach, the different packages that comprise the ICUBoot have been defined by adding the corresponding models without having to

modify their source code. Also, the software interface models have been defined. Figure 7 shows the software packages and software interfaces of the ICUBoot, including the previously described `ccsds_protocol` package.

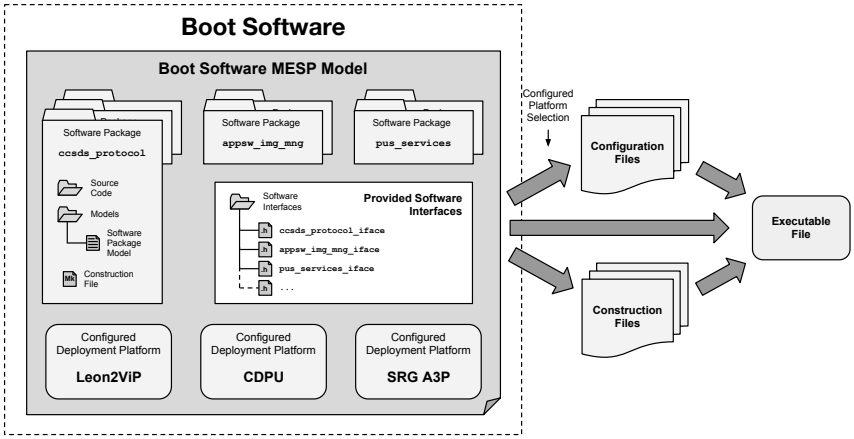


Figure 7: ICUBoot MESP model schema.

Once the different packages are created, the complete ICUBoot can be defined using a model called Multi-platform Embedded Software Project (MESP), which includes all the software packages that are part of it. Apart from that, the model incorporates the vectors of parameter values used to configure the packages and the underlying deployment platform. The MESP model allows developers to incorporate in a single model different deployment alternatives, i.e., combinations of software packages; and different deployment platforms. Like this, one single model can seamlessly follow the whole development process by including all the platforms and software package combinations used during the different development stages (e.g. engineering models, qualification models, etc.), version-tagging and storing in each case the configuration parameters used. From an application's MESP model, and once the targeted deployment alternative and platform have been chosen, the construction and configuration files of the application can be generated, which can subsequently be used to build the final executable file.

The MESP model defined for the ICUBoot defines three different configured deployment platforms: `SRG A3P`, `Leon2ViP` and `CDPU`. The first one is

used to deploy applications on the SRG's A3P development board, used during the first stages of the development process. The second one corresponds to the Leon2ViP virtual platform, whose main characteristics are described in Subsection 3.2. Finally, the last one is the configured deployment platform corresponding to the final CDPU flight model. Figure 7 shows an usage schema of the MESP model of the ICUBoot. From the model, and incorporating the information of the selected software packages, MICOBS is capable of generating the configuration and construction files needed to build the final binary file. Following the platform-aware approach, the generative process is triggered from the MESP model by the selection of the targeted deployment platform that will execute the executable image.

3.2. ICU Application Software

This software is in charge of the ICU when working in nominal mode. It controls the communication among the sensors and the ICU, and also between the spacecraft and the EPD instrument. It checks the data received from the sensors, forwarding the science telemetry to the spacecraft. It also receives from the spacecraft the EPD telecommands, executing those that are addressed to the ICU and rerouting the rest to the corresponding sensor. Finally, it provides the subset of PUS services required for Solar Orbiter payloads, such as housekeeping, on-board monitoring, event reporting, memory and time management or telecommand verification.

In nominal mode, scientific data from the sensors are gathered and timestamped periodically every second. They are stored in a ring buffer with their corresponding time-tags. Once stored, two scenarios arise depending on whether the burst mode is triggered or not. Due to the low bandwidth of the EPD instrument, i.e. 3100 bps, in normal mode of operation the time resolution of the EPD scientific data is reduced by a factor of 10. This means that 10 seconds of data with a resolution of one second are compressed into a single data bin with a resolution of 10 seconds. This operation basically implies calculating the mean value of the 10 one-second-resolution samples. Only when the burst mode is triggered, time resolution is kept to one second. Due to the fact that the triggering of the burst mode can be received with some delay, specially when other instruments trigger EPD, it is necessary to store in a ring buffer the timestamped data received from the sensors. Thus, when the trigger arrives, the application software can go "back in time" and send the stored data with one second resolution. If no trigger is received, data binning is applied and the data is sent to the spacecraft with 10 seconds

resolution. Approximately 10% of the returned data is expected to be in burst mode, with ≈10 times the data rate of normal mode, corresponding to ≈1% of the time.

The ICUAppSw is organized in different layers, as shown in Figure 8. Each layer is independent from the rest, and the exchange of information among them is done through well-defined interfaces.

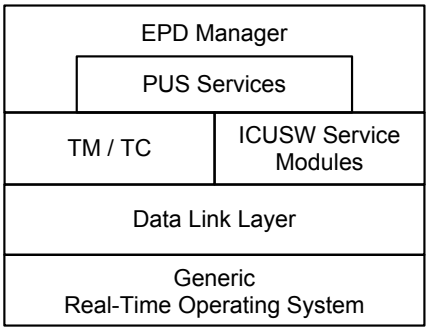


Figure 8: ICU Application Software layers.

The bottom layer corresponds to the Real-Time Operating System (RTOS). This layer is in charge of thread management, scheduling, timing and synchronization. Using an approach equivalent to Operating System Abstraction Layer (OSAL) (Oliver et al., 2010), this layer has been designed as a generic RTOS abstraction layer that wraps any concrete implementation, so different platforms with different RTOS can be used. Platforms based on the EDISOFT version of RTEMS (Silva et al., 2009) and Linux have been constructed. RTEMS 4.6 is the RTOS selected for the final flight configuration, while Linux is used in integration and unitary tests scenarios.

The Data Link layer provides to the upper software layers an abstraction of the UART/LVDS links with the sensors and the SpaceWire link with the spacecraft. After its integration in MICOBS, this layer can be also configured to allow the substitution of physical links by virtual links associated to files. This configuration is used on the Linux platform for developing unitary and integration test cases on a simple PC. This approach provides two significant advantages: firstly, an early validation process can be performed, even if the real EPD hardware elements or their emulators are not available; secondly, the analysis of the software response under unlikely scenarios can be easily

constructed using files instead of real hardware systems.

In order to manage the deployment of the unitary and integration tests on the different platforms, MICOBS software packages and interfaces have been defined for every layer of the ICUAppSw. Thus, the same platform-aware approach used in the development of the ICUBoot is followed. Each one of the tests will be managed by a specific MESP model supported by MICOBS similar to the one shown in Figure 7.

The TM/TC layer controls the routing of telemetry and telecommand traffic among the spacecraft, the sensors and the ICU. The PUS Services layer implements the whole set of PUS services (ECSS Secretariat, 2003a) that the instrument must provide. The ICUSW Service Module, meanwhile, provides a broad set of services, such as telemetry formatting, controlled access to ICUSW data, software module patching management, etc.

Finally, the upper layer, called EPD Manager, defines the top-level software entities of the ICUAppSw that cooperate for configuring, controlling and monitoring both the EPD sensors and the ICU itself. To develop this layer, and to manage the complete deployment of the ICUAppSw, graphical modelling and automatic code generation techniques have been combined with the platform-aware approach and the support for component-based system construction and deployment provided by MICOBS. This component approach and its benefits have been tested in previous satellite missions (Polo et al., 2012) and it is sustained on a component design CASE tool, called EDROOM (Polo et al., 2001). The following subsection describes in detail the construction and deployment process of the ICUAppSw and the role played in it of the EDROOM and MICOBS tools.

3.2.1. ICUAppSw component-based construction and deployment

The EPD Manager layer includes the top-level entities assigned to the software after the partitioning. These entities have been implemented as EDROOM components. EDROOM provides for that a graphical editor for UML2 components and an automatic Embedded C++ code generator.

The overall composition structure of this layer is depicted in Figure 9. `EPDManager` is the main component, which contains three sub-component instances: `HK_FDIRManager`, `SensorTMManager` and `BackGroundTCExecutor`. A shared resource called `SCTxChannelCtrl` works as a stand-alone component that is accessible from the rest of components. Figure 9 shows the subscriptions of the components to the timing and interrupt/exception services, as well as their communication topology. Each subscription is done by

instantiating the corresponding port through which the messages triggered by the timeout, exception and interrupt events will be received. This graphical representation, together with the triggering pattern (periodic or sporadic) of each one of the events, define the real-time design of the system.

The role of each one of these components is detailed below:

`EPDManager`: this component periodically retrieves in a polling mode the telecommands addressed to the ICU from the spacecraft, and it executes the priority ones. The non-priority telecommands are forwarded to other components, depending on their related service. It also manages the EPD modes and the EPD critical events, including those related to software exceptions of the ICU.

`HK_FDIRManager`: this component periodically performs the actions concerning housekeeping and fault detection, isolation and recovery (FDIR). It is responsible of ensuring that critical events detected during these actions are notified to the `EPDManager`. Finally, it also executes the housekeeping service telecommands forwarded by the `EPDManager`.

`SensorsTMManager`: this component periodically retrieves the EPD sensors' telemetry, making sure that the science data baud-rate is under the specified limits. It also detects and notifies the `EPDManager` the critical events associated to the received telemetry. Finally, it executes the Science service telecommands forwarded by the `EPDManager`.

`BKGTCExecutor`: executes the background telecommands received from the `EPDManager`.

`SCTxChannelCtrl`: is a shared resource that controls the transmission buffer of PUS packets from the ICU to the spacecraft. The rest of components access this resource in order to enqueue the telemetry packets that must be sent to the spacecraft. Internally, this resource is implemented as a queue of telemetry descriptors pending to be sent and whose access in mutual exclusion is controlled by a semaphore with a priority ceiling policy. The transmission of the queued packets is later handled by dedicated hardware.

The behaviour of each component has been defined using a formalism called ROOMCharts (Selic et al., 1994). EDROOM provides a graphical editor for ROOMCharts and automatically generates Embedded C++ code.

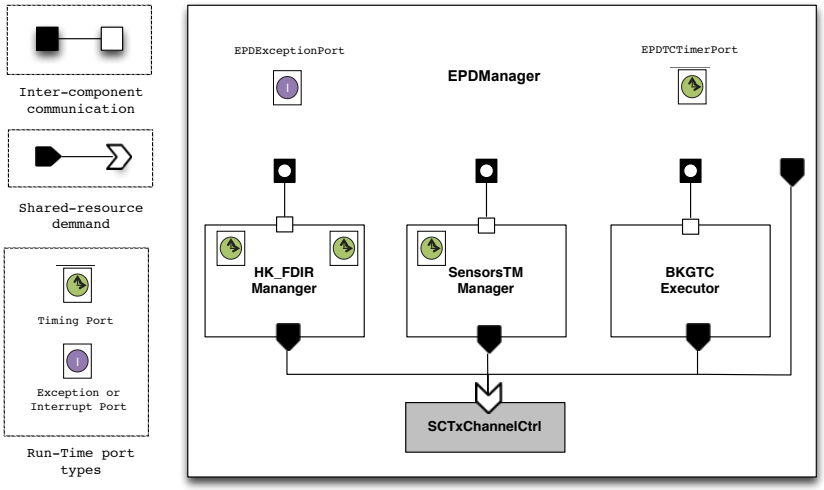


Figure 9: ICUSW composition structure.

Figure 10 shows the behavior of the `EPDManager` component using this formalism. Each transition is triggered by a message reception and it has associated a message handler action. Choice points, as those associated to the signals *TCRetrieveTimeOut* and *CriticalEvent*, allows the definition of different behavioural scenarios after the data attached to the corresponding received message have been processed. Specifically, the branches of *TCRetrieveTimeOut* manage all the possible types of telecommands that EPD can receive from the spacecraft. EDROOM generates platform independent code for each component, which is later integrated in MICOBS for the construction of the ICUAppSw.

This integration is performed using the composition level of the MICOBS framework. This level, logically located above the previously described packaging level, allows the description of the high-level component architecture of an application and the generation of different products using a model-driven engineering approach. In order to do so, the framework defines an artifact, called *component domain* with which component technologies, such as EDROOM, can be incorporated into the framework. Once a technology has been integrated, components can be imported, and applications can

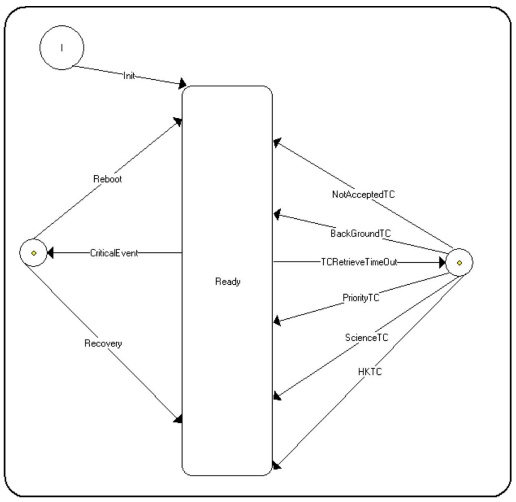


Figure 10: `EPDManager` component behaviour definition.

be defined using the MICOBS architectural model of an application, called Multi-platform Component Architecture Description model or MCAD. This model includes the set of component instances that are part of the application, the connections established among them and the set of *service libraries*. These libraries represent passive modules that are compiled and linked together with the components and that provide them with different supporting functions. In the case of the ICUAppSw, these libraries correspond to the lower level layers, i.e., PUS services, TM/TC, ICUSW service modules and data link layers. The lowest level layer, this is, the RTOS, is not defined as a service library since it is included as a constituent part of the deployment platform.

The MCAD model includes the capability of defining multiple alternatives and multiple configured deployment platforms as its packaging model counterpart.

As it has been previously mentioned, a component domain can define a set of transformations that allow the developers to obtain different products from the MCAD model of an application, which can be, for example, the final executable of the application. EDROOM has been fully integrated in MICOBS. The corresponding domain includes one single type

of component, i.e., `EDROOMComponent`, one type of component port, namely `EDROOMPort`, one type of interface called `EDROOMProtocol` and one connector, `EDROOMConnector`. Listing 2 shows the model of the `EPDManager`. This model includes the languages in which the component is implemented, the services it requires that will be eventually provided by the deployed service libraries, the platforms it supports (in this case, the component platform independent), and its internal structure in the forms of internal ports and instances and the connections among them. Only the architectural ports are described in the model, i.e., internal service ports, such as the exception and timing ports of the `EPDManager`, are not included since they are not part of the external interface of the component.

The EDROOM domain defines a set of transformations with which, the developers are capable of generating the glue-code of an application, this is, the code that instantiates and connects all the component instances, and the equivalent MESP model. From this MESP model, the framework can generate, as explained in the ICUBoot development description, the construction and configuration files that will lead to the obtention of the final executable file.

Apart from all the aforementioned functionality, MICOBS allows the integration of different analysis tools into the framework. In this way, components can be annotated using specific models in order to obtain reports from the analysis of system properties using the composability and compositionality principles (Gössler and Sifakis, 2005). Thanks to the platform concept included in the framework, the annotations can be defined in terms of the platforms on which the components can be deployed. From the MCAD models of a system, and using the information attached to the components and service libraries that part of it, products can be obtained that are later used as inputs for tools that perform analysis on given properties. Currently, the MAST (Harbour et al., 2001) tool has been integrated into MICOBS in order to perform the schedulability analysis of the ICUAppSw (Fernández et al., 2013).

Figure 11 shows an usage schema of the MCAD model of the ICUAppSw. This model includes the same deployment platforms that were used in the ICUBoot software development plus one corresponding to a PC/Linux hardware used to perform functional tests. In this case, two deployment alternatives are defined. Even though the component architecture is the same for both alternatives, each of them deploys a specific set of service libraries. When the targeted platform is a simulated one, i.e. it is the Leon2ViP or

Listing 2 `EPDManager` component description model.

```
component EDROOM(v1)::EDROOMComponent epd_manager
{
 version := v1;
 languages := CPP(98);
 requires := cdtcdescriptor(v1), cdepdevent(v1);
 supported platforms {
   supported platform any_any_any_any_any {
     osapi := any;
     os := any;
     architecture := any;
     microprocessor := any;
     board := any;
   };
 };
 internal ports {
   client EDROOM(v1)::EDROOMPort CP_TPort(v1) hk_fdir { };
   client EDROOM(v1)::EDROOMPort CP_TPort(v1) bg_tc { };
   client EDROOM(v1)::EDROOMPort CP_TPort(v1) sensor_tm { };
 };
 subcomponent instances {
   instance ca_hk_fdir_mng(v1) hk_fdir_mng { };
   instance ca_bg_tc_exec(v1) bg_tc_exec { };
   instance ca_stm_mng(v1) stm_mng { };
 };
 connections {
   connection this.hk_fdir <-> hk_fdir_mng.hk_fdir
     using EDROOM(v1)::EDROOMConnector {};
   connection this.bg_tc <-> bg_tc_exec.bg_tc
     using EDROOM(v1)::EDROOMConnector {};
   connection this.sensor_tm <-> stm_mng.sensor_tm
     using EDROOM(v1)::EDROOMConnector {};
 };
};
```

the PC/Linux, the drivers of the ICU hardware interfaces are replaced by a set of routines that will retrieve the input data from local files rather than from the actual communication bus.

Taking the model as input, the EDROOM domain transformations generate the equivalent MESP model and the application's glue-code. From this, the construction and configuration files can be obtained in a way identical to that of the ICUBoot. Apart from the MESP model, the models that will serve as inputs for the analysis tools, as MAST, can also be obtained. In both cases, the generate process is platform-driven, i.e., the targeted deployment platform shall be selected before the transformations are triggered.

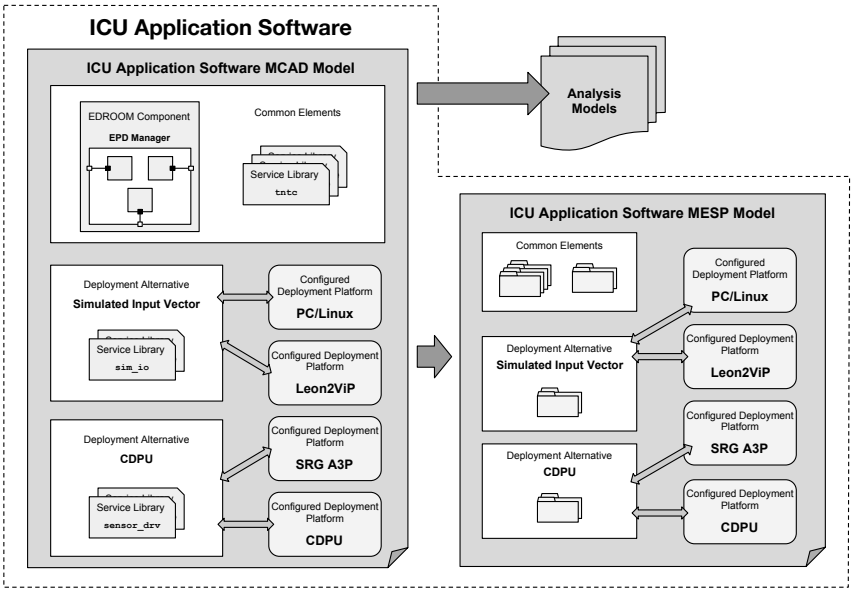


Figure 11: ICUAppSw MCAD model usage schema.

3.2.2. Fault injection and HW/SW co-simulation platform

As it was previously mentioned, in order to fulfill the fault tolerance requirements, the ICUBoot has to perform several complex sanity checks of ICUAppSw stored binaries and SDRAM runtime areas before the final application deployment. Since in this early boot stage there are no software

services at all, it is hard to accomplish a complete verification of fault tolerant requirements as are specified in the EPD Flight Software Requirements Document (EPD Team, 2012) on real hardware. These robustness requirements call for an exhaustive testing of the ICUBoot functionality against a possible corruption of the ICUAppSw application binaries stored in the EEPROM or stuck-at faults in the SDRAM application deployment areas or in the area where the ICUBoot stack can be located. This asks for a specific fault injection framework capable of emulating memory permanent errors.

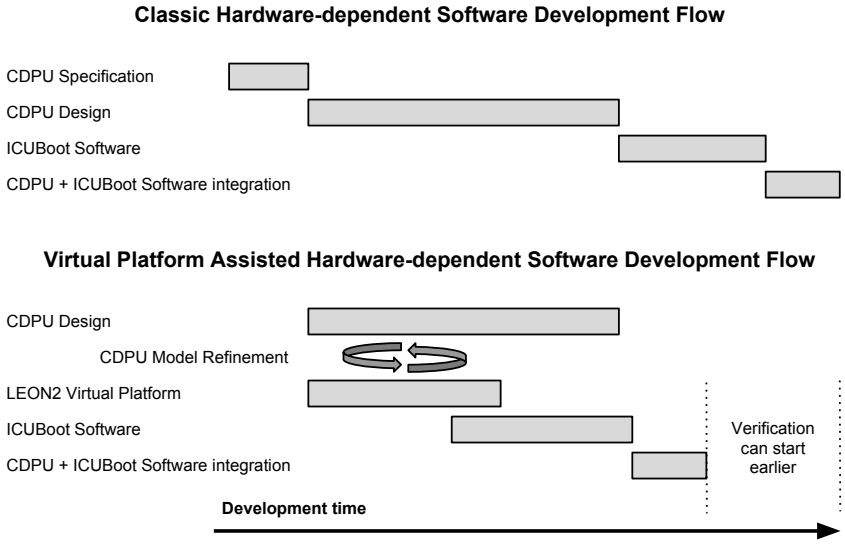


Figure 12: Improvement of the software development flow.

Other issues that have to be beard in mind are the effort and risk associated with integrating hardware dependant software, like ICUBoot. The classic approach to develop this kind of software is to design the hardware, make a physical prototype, write the code, and then integrate the hardware and software. This methodology is nowadays too slow and asks for an alternative to the traditional "software-after-hardware" design flow in order to get started with software before the hardware is ready. To shortcut these issues, the SRG has developed a LEON2 virtual platform (Leon2ViP) with

fault injection capabilities. This way, it is possible to run the same binary software on the simulated target as it does on the physical system, but in a completely controlled environment, allowing earlier ICUBoot development and stricter requirements verification.

Leon2ViP has been coded using a loosely-timed coding style. This means that its main use is intended for software development from a functional point of view. Real-time analysis like tasks scheduling or worst case execution are not covered in this release. However, although the instruction set simulator runs free at the speed of the host where the virtual platform is running on, there is a real time synchronization point defined by the operating system clock tick. Since Leon2ViP is timer tick accurate, it is possible to run preemptive multitasking embedded operating systems based on time slice scheduling like RTEMS or eCos. Figure 13 shows the execution of an eCos operating system application and reflect the virtual platform capability to execute this kind of multitasking applications on top of embedded operating systems.

Another important point to note is the virtual platform capability to carry out the design space exploration of custom IP Cores software interfaces and, in an incremental way, to advance in the IP Cores refinement until a fully functional hardware-software co-simulation is feasible. Having the memory mapping of a hardware register set in TLM2.0 with a dummy behaviour implementation is an adequate starting point for device drivers or low level software libraries development when the specific hardware is not yet available. The behavior and the interface can be refined while the IP Core is developed separately. As an example, this approach allowed the co-design of SpaceWire services needed by the ICUboot and the later hardware-software co-simulation, once the functionality of the SpaceWire IP Core was integrated in Leon2ViP.

For the ICUBoot and ICUAppSw development, Leon2ViP provides faster edit, compile and debug cycles (see Figure 12) and, at the same time, a more controllable and observable environment for the verification activities. Furthermore, even if the hardware was already available, this virtual platform offers, not possible otherwise, non-intrusive and exhaustive debug and fault injection capabilities and the co-simulation with a fully functional version of the SpaceWire IP Core. The platform also allows the developers to work with hardware-in-the-loop, like a real SpaceWire controller connected to the host that interfaces with other devices, such as spacecraft simulators.

This LEON2 virtual platform is a transaction-level model of a complete

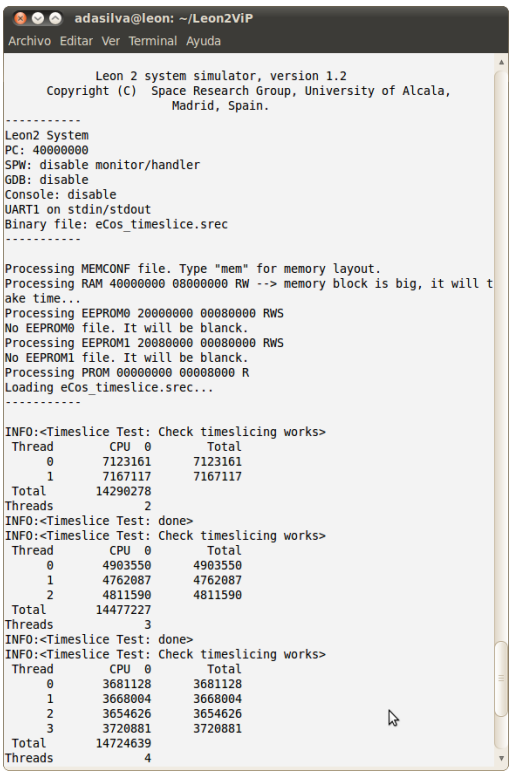


Figure 13: Leon2ViP eCos execution.

LEON2 system. It has been developed using SystemC/TLM2.0 interfaces as shown in Figure 14. Its main components are:

- **LEON2 ISS**: is a SPARC V8 untimed Instruction Set Simulator (ISS) with a blocking TLM2 transaction interface.
- **Memory**: includes PROM, EEPROM and SDRAM blocks, as were described in Section 2.1. The current contents are read from external ordinary binary images or Executable and Linkable Format (ELF) files generated by the compiler toolchain.

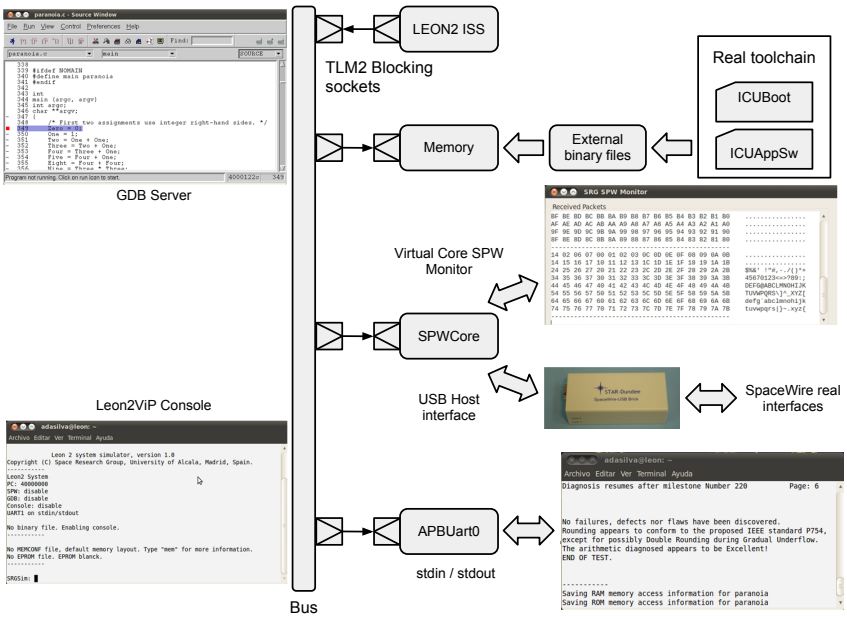


Figure 14: LEON2 ViP transaction model.

- **SPWCore**: implements a SpaceWire interface for on-board communications. This interface can be mapped to a SpaceWire monitor or to an external SpaceWire real hardware, like a STAR-Dundee USB SpaceWire brick[1].

- **APBUart0**: is an interface provided for serial communications. It is used by the software to perform standard input/output operations and can also be mapped to a real host serial port.

For controlling the execution of the applications, Leon2ViP provides a command-line interface or shell. This shell allows the user to issue several commands like program load, set/unset breakpoints and watchpoints, etc.

[1] http://star-dundee.com/products/spacewire-usb-brick

A GDB server is also included, which is able to operate through a TCP/IP network connection.

In order to verify the correctness of Leon2ViP virtual platform implementation several tests have been conducted. One of them is the Stanford benchmark, delivered along with the `sparc-elf-gcc` toolchain used to generate the executables and provided by Gaisler Research (Aeroflex Gaisler, 2010). All Leon2ViP tests were carried out using a 1,66 Ghz generic laptop with 1 Gbyte RAM Memory under Ubuntu 12.10. The results provided by Leon2ViP virtual platform are compared to a real LEON2 deployed on an FPGA A3P board developed by SRG group. The system runs at 20Mhz, which provides 17 MIPS performance. This clock frequency is the same that is going to be used in the ICU final system processor board.

As is expected Leon2ViP runs faster than A3P system. Depending on the benchmark Leon2ViP is 1.5 to 5 times faster. But from a functional point of view, the benchmark worth is not so much the speed of the executed tests, but rather the accuracy of the final results obtained. By knowing the expected outcome it is possible to establish the correctness of the execution. This gives great confidence in the Leon2ViP implementation. From a functional point of view, no discrepancies were found. Stanford results are shown in Figure 15.

4. Related works

There are similar approaches to the one described in this work. TASTE (Perrotin et al., 2012) is an open source environment intended for the development of embedded real-time systems. It is the continuation of the project ASSERT (Hugues et al., 2008a). The environment is capable of combining heterogeneous components that can be written in different languages, including Matlab or VHDL in the latest version. In order to do so, it defines a common interface representation and uses the ASN.1 language (Mamais et al., 2012) to describe the different types of data exchanged between the components. From the multi-view definition of a system, the tools are capable of obtaining a common AADL language representation from which different products can be obtained, including the skeleton code of the application using Ocarina (Hugues et al., 2008b), which follows the Ravenscar Computational Model. The generated code, which can be either Ada or C, is executed on top of the PolyORB-HI middleware (Vergnaud et al., 2004). Furthermore, it integrates Cheddar (Singhoff et al., 2004) and MAST anal-

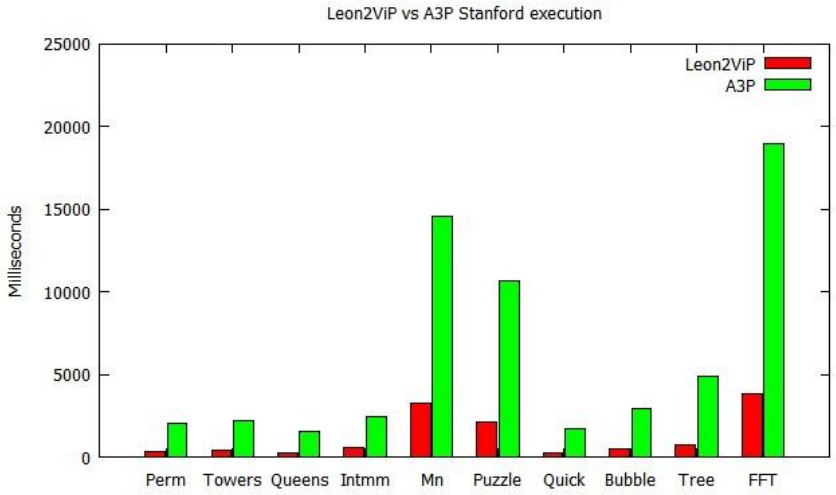


Figure 15: Leon2ViP vs A3P board Stanford test.

ysis tools. One of the limitations of TASTE is that it forces you to use its component model, with its underlying technology to benefit from the functionality it offers (model-checking, analysis, etc.). MICOBS, thanks to the component domains, is capable of integrating different technologies and to reuse the models and generators that are part of the integration of the analysis tools. Furthermore, the use of the ASN.1 language and its associated codecs introduces a certain degree of overhead.

The existing analysis tools, such as MAST and Palladio, as well as the TASTE environment, lack of a proper annotation mechanism to allow the developers to assign values to the different element properties depending on the platform on which they can be deployed. MICOBS provides this functionality thanks to the platform model it incorporates and the platform-aware approach that drives the model and process definitions.

The Deployment and Configuration (D&C) specification of the Object Management Group (OMG) (OMG, 2006b) is intended for performing the deployment and configuration of component-based applications. It is mainly focused in distributed environments and specifically designed for applications implemented using the CORBA Component Model (CCM) (OMG, 2006a).

Originally, the different models included in the specification do not include by default any means for annotating the different elements, although works have been done in this direction (López Martínez et al., 2008). The D&C is complex and includes artifacts, such as components, distributed nodes and connections, which are not suited for defining component-technology-less embedded applications such as the ICUBoot.

Regarding the use of virtual platforms, SystemC/TLM2 is an interoperability standard for memory-mapped bus modeling and is a key enabler for the development of virtual platforms, serving as a bridge between hardware and embedded software designers, specially for hardware dependant and communication software development. Virtual platforms have become widely used in design space exploration and early software development in avionics and space software environments, before the hardware becomes available (Randimbivololona et al., 2012; Wang et al., 2012).

5. Conclusions

The Energetic Particle Detector instrument, part of the Solar Orbiter payload, requires a sophisticated Instrument Control Unit to operate the suite of five different sensors that are part of it. The ICU is able to collect and monitor the data provided by the sensor and defined in the scope of the Solar Orbiter mission, which are necessary to fulfil the scientific objectives. The ICU routes commands to the sensors, controls the power system, collects instrument housekeeping and directs science data to the spacecraft's on-board mass memory.

A hardware/software approach has been followed in the development of the ICU. After a trade-off analysis, where the FPGA resource limitations and the impact of the key requirements in the software complexity have been taken into account, a final partitioning decision has been made. The resulting hardware has thus been built by a combination of third-party and *ad-hoc* IP Cores together with the LEON2 processor sharing the AMBA bus.

In turn, the software has been developed using MICOBS, a platform-aware development framework that facilitates the control over the deployment of the software on the different platforms used during the whole design cycle. These platforms include the SRG A3P breadboard, the ICU flight model and the Leon2ViP virtual platform. Leon2ViP enables unmanned and tightly focused fault injection campaigns, not possible otherwise, in order to early expose and diagnose flaws in the software implementation.

In this project, the use of Leon2ViP has being used to check the correct behaviour of the boot software in the face of possible stuck-at faults in the SDRAM or in the EEPROM that stores the application software binaries. The virtual platform has also allowed a hardware-software co-simulation, incorporating the SpaceWire IP Core and checking its correct integration with the ICU software.

Together with MICOBS, the top-level entities of the on-board software has been developed using the EDROOM graphical modelling and code generation tool. This tool is capable of generating from their behavioural description, the code of the different components that comprise the application. Thanks to the integration of the EDROOM domain into MICOBS, the defined components can be imported and the architectural description model of the application software can be defined. From this later model, and using model-driven engineering techniques, the developers are able to manage the deployment of the application on different platforms and, moreover, to obtain different products. These products include the input model of the MAST tool, with which the software schedulability analysis is performed.

Finally, it must be emphasized that all the development tools used have been developed by the Space Research Group, which paves the way for a continuous improvement of the whole process. In this respect, nearest future works include the integration of more IP Cores into the Leon2ViP environment so that they can be part of the co-simulation tests.

It must be emphasized that all the development tools used have been developed by the SRG, which paves the way for a continuous improvement of the whole process. In this respect, nearest future works include the integration of more IP Cores into the Leon2ViP environment so that they can be part of the co-simulation tests.

References

Aeroflex Gaisler, October 2010. Rcc user's manual. Available at http://www.gaisler.com/anonftp/rcc/doc/rcc-1.2.0.pdf.

ARM, 1999. AMBA specification, revision 2.0.

CCSDS, 2000. Packet telemetry, CCSDS 102.0-b-5, blue book. Tech. rep., CCSDS.

da Silva, A., Sánchez, S., 2010. LEON3 ViP: A virtual platform with fault injection capabilities. In: Proceedings of the 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools. DSD '10. IEEE Computer Society, Washington, DC, USA, pp. 813–816.

ECSS Secretariat, 2003a. Ground systems and operations — telemetry and telecommand packet utilization. Tech. rep., ESA-ESTEC.

ECSS Secretariat, 2003b. Spacewire - links, nodes, routers and networks, ecss-e-50-12a. Tech. rep., ESA-ESTEC.

EPD Team, 2012. EPD software requirements document, SO-EPD-ICU-RS-0002. Tech. rep., University of Alcala.

ESA, 2012. ESA's cosmic vision. Available at http://www.esa.int/esaSC/SEMA7J2IU7E_index_0.html.

Fernández, J., Parra, P., García, I., Sánchez, S., Óscar R. Polo, 2013. Schedulability analysis of on-board satellite software based on model-driven and compositionality techniques. In: IEEE International Symposium on Industrial Embedded Systems. To appear.

Gaisler Research, 2005. LEON2 processor user's manual, version 1.0.30. Tech. rep., Gaisler Research.

Gössler, G., Sifakis, J., March 2005. Composition for component-based modeling. Sci. Comput. Program. 55, 161–183.
URL http://portal.acm.org/citation.cfm?id=1065095.1065101

Harbour, M. G., García, J. J. G., Gutiérrez, J. C. P., Moyano, J. M. D., 2001. MAST: Modeling and analysis suite for real time applications. In: Proceedings of the 13th Euromicro Conference on Real-Time Systems. ECRTS '01. IEEE Computer Society, Washington, DC, USA, pp. 125–.

Hugues, J., Perrotin, M., Tsiodras, T., 2008a. Using MDE for the rapid prototyping of space critical systems. In: Proceedings of the 2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping. IEEE Computer Society, Washington, DC, USA, pp. 10–16.
URL http://portal.acm.org/citation.cfm?id=1447559.1447631

Hugues, J., Zalila, B., Pautet, L., Kordon, F., August 2008b. From the prototype to the final embedded system using the Ocarina AADL tool suite. ACM Trans. Embed. Comput. Syst. 7, 42:1–42:25.
URL http://doi.acm.org/10.1145/1376804.1376810

JSTDT, 2008. Heliophysical Explorers (HELEX): Solar Orbiter and Sentinels. Tech. rep., NASA.

López Martínez, P., Drake, J. M., Pacheco, P., Medina, J. L., 2008. Ada-CCM: Component-based technology for distributed real-time systems. In: Proceedings of the 11th International Symposium on Component-Based Software Engineering. CBSE '08. Springer-Verlag, pp. 334–350.

Mamais, G., Tsiodras, T., Lesens, D., Perrotin, M., 2012. An ASN.1 compiler for embedded/space systems. In: ERTS 2012.

Oliver, R. S., Shcherbakov, I., Fohler, G., 2010. An operating system abstraction layer for portable applications in wireless sensor networks. In: Proceedings of the 2010 ACM Symposium on Applied Computing. SAC '10. ACM, New York, NY, USA, pp. 742–748.

OMG, 2006a. CORBA component model specification v4.0. Available at http://ww.omg.org/docs/formal/06-04-01.pdf.

OMG, 2006b. Deployment and configuration of component-based distributed applications specification v4.0. Available at http://ww.omg.org/docs/formal/06-04-02.pdf.

Parra, P., Polo, O. R., Knoblauch, M., García, I., Sánchez, S., 2011. MICOBS: multi-platform multi-model component based software development framework. In: Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering. CBSE '11. ACM, New York, NY, USA, pp. 1–10.

Perrotin, M., Conquet, E., Delange, J., Schiele, A., Tsiodras, T., 2012. TASTE: A real-time software engineering tool-chain overview, status, and future. In: SDL 2011: Integrating System and Software Modeling. Springer Berlin Heidelberg.

Polo, O. R., de la Cruz, J. M., Girón-Sierra, J., Esteban, S., 2001. EDROOM. automatic C++ code generator for real-time systems modelled with ROOM. In: NTC2001 IFAC Conference.

Polo, O. R., Parra, P., Knoblauch, M., García, I., Sánchez, S., Angulo, M., 2012. Component-based engineering and multi-platform deployment for nanosatellite on-board software. In: Proceedings of the DASIA 2012 Conference.

Randimbivololona, F., Brahmi, A., Meur, P. L., 2012. Airborne software tests on a fully virtual platform. CoRR abs/1204.3410.

Selic, B., Gullekson, G., Ward, P. T., 1994. Real-time object-oriented modeling. John Wiley & Sons, Inc., New York, NY, USA.

Silva, H., Sousa, J., Freitas, D., Faustino, S., Constantino, A., Coutinho, M., 2009. RTEMS improvement - space qualification of RTEMS executive. In: Proceedings of the 2009 INFORUM Symposium.

Singhoff, F., Legrand, J., Nana, L., Marcé, L., 2004. Cheddar: a flexible real time scheduling framework. In: Proceedings of the 2004 annual ACM SIGAda international conference on Ada: The engineering of correct and reliable software for real-time & distributed systems using Ada and related technologies. SIGAda '04. ACM, New York, NY, USA, pp. 1–8.
URL http://doi.acm.org/10.1145/1032297.1032298

Vergnaud, T., Hugues, J., Pautet, L., Kordon, F., 2004. PolyORB: A schizophrenic middleware to build versatile reliable distributed applications. In: Ada-Europe. pp. 106–119.

Wang, Y., Wang, L., Zheng, Z., 2012. Application of virtual prototype technology to simulation test for airborne software system. In: Jin, D., Lin, S. (Eds.), Advances in Electronic Engineering, Communication and Management Vol.2. Vol. 140 of Lecture Notes in Electrical Engineering. Springer Berlin Heidelberg, pp. 653–658.
URL http://dx.doi.org/10.1007/978-3-642-27296-7_100

A.4. Runtime instrumentation of SystemC/TLM2 interfaces for fault tolerance requirements verification in software cosimulation

Autor

Antonio da Silva, Pablo Parra, Óscar R. Polo y Sebastián Sánchez

Lugar

MODELLING & SIMULATION IN ENGINEERING, ISSN: 1687-5591, vol. 2014, Article ID 105051

Tipo

Journal paper

Ref

SCImago 2013: Category Modelling and Simulation Q4

Resumen

Este artículo describe el diseño de una librería para la interceptación de llamadas en la interfaz TLM2.0. Esta librería puede usarse para la comprobación/aserción de propiedades del sistema, verificación del protocolo de llamadas o inyección de fallos. La librería usa modificaciones de la tabla de métodos virtuales en C++ como técnica de instrumentación dinámica de código para interceptar llamadas en la interfaz TLM2.0. Esta técnica puede ser usada después de la fase de elaboración del sistema en SystemC y no necesita modificaciones del código fuente ni recompilación de los módulos para la inserción de "wrappers" en las interfaces. La técnica propuesta ha sido aplicada con éxito en la verificación del software de arranque de la unidad de control del instrumento (ICU) detector de partículas energéticas a bordo de Solar Orbiter.

Runtime instrumentation of SystemC/TLM2 interfaces for fault tolerance requirements verification in software cosimulation

Antonio da Silva[1,*], Pablo Parra[2], Óscar R. Polo[2], Sebastián Sánchez[2]
1 ETSIS Telecomunicación, Universidad Politécnica de Madrid, Madrid, Spain
2 Computer Engineering Department, Universidad de Alcalá, Alcalá de Henares, Madrid, Spain
∗ **Corresponding author: antonio.dasilva@upm.es**

Abstract

This paper presents the design of a SystemC Transaction Level Modelling wrapping library that can be used for the assertion of system properties, protocol compliance or fault injection. The library uses C++ Virtual Table hooks as a Dynamic Binary Instrumentation technique to inline wrappers in the TLM2 transaction path. This technique can be applied after the elaboration phase and needs neither source code modifications nor recompilation of the top level SystemC modules. The proposed technique has been successfully applied to the robustness verification of the on-board boot software of the Instrument Control Unit of the Solar Orbiter's Energetic Particle Detector.

1 Introduction

Embedded software plays an important role in today's complex System-on-Chips (SoCs) since it allows convoluted features to be implemented flexibly. However in new developments, hardware capable of executing software is not often available until the later stages of the development cycle. To accelerate the design process and to increase the productivity, system-level design has to accommodate software concerns enabling a seamless co-design of software and hardware. Therefore, it is highly desirable to address software development as early as possible. Virtual platforms are executable models of complete systems that provide software developers with working frameworks time before the real hardware is available. They enable the concurrent development of System-on-Chip (SoC) hardware and software, significantly shortening their integration times. From an embedded software perspective, the use of virtual platforms allows the development and verification processes to be started earlier in the design flow so as to detect and correct errors that would otherwise propagate towards the final implementation stages. Moreover, it is easier to access and modify the internal state of the virtual prototypes, so that a comprehensive fault injection campaign and fault tolerance assessment can be carried out. This helps to achieve what every embedded software developer is fundamentally looking for; predictability and robustness [18].

System-level modeling languages are used to start the design process from an abstract level and apply a top-down design methodology through a refinement process [29]. The SystemC Transaction Level Modeling [27] raises the abstraction level of system descriptions, focusing on the exchange of data between components through communication channels or sockets. Because of their advantages, the descriptions of TLM systems can be used for design space exploration, early architectural performance estimations and to allow an earlier software development commencement by joining the hardware and software design flow together. SystemC/TLM is written in C++, which is a widely known programming language that has been a popular starting point for describing executable hardware/software systems models. Using TLM, system models are quick to write and give an executable version of the specification, which allows a very fast simulation. For system-level design, these languages allow hardware and software components to be described in a single framework. Furthermore, the only development tools needed are regular C/C++ compilers and debuggers, with which embedded systems designers are already well acquainted. The use of virtual platforms gives developers far more visibility and control over system design in comparison to traditional development methodologies. Any state is within reach and any

condition can be triggered. Therefore, virtual platforms have become widely used in avionics and space software development environments before the hardware becomes available. Current research focuses on experimental techniques and tools that allow software robustness verification through fault injection using virtual platforms.

The remainder of the paper is organized as follows: Solar Orbiter's mission characteristics along with the embedded software development challenges and novel contributions are set out in section 2, relevant related works are detailed in the same section. Section 3 describes the proposed wrapper insertion framework. Section 4 discusses some issues as regards performance and usability using different interposition code insertion techniques. Section 5 describes the real scenario in which an early software robustness evaluation by means of fault injection on a TLM2 virtual platform (Leon2ViP) has been carried out. Finally, section 6 contains the conclusions.

2 Problem Statement and related work

2.1 Solar Orbiter Mission

Solar Orbiter [13] is a planned Sun-observing satellite, under development by ESA and is scheduled to be launched in January 2017 as a baseline. At its closest point, the spacecraft will be closer to the Sun than any previous spacecraft, almost one-third of the Earth's distance from the Sun. Because of the proximity of the Sun, the spacecraft must withstand powerful bursts of atomic particles coming from the solar atmosphere. From an on-board software designer's perspective, it is essential to look out for permanent soft errors resulting from latch-up failures in SDRAM/EEPROM memories. The Space Research Group (SRG) of the University of Alcalá is in charge of the development of the Instrument Control Unit (ICU) for the Energetic Particle Detector on-board Solar Orbiter along with the corresponding boot and application software. For the early development and verification of the ICU's bootloader software, a framework with the ability to run the same binary code that will run on real hardware was needed. It also had to emulate SDRAM and EEPROM permanent errors, a fact that is difficult, if not impossible, on real hardware. The ICU boot software is in the critical path of the project so its verification should be addressed at an early development stage, for any test case missed in this process can affect the quality of the overall onboard software. Thus, the robustness requirements call for an exhaustive testing of the boot process and possible corruption of application binaries stored in the EEPROM or stuck-at faults in the SDRAM application deployment areas. Bearing this in mind, from a hardware dependent software point of view, such as the boot software, the major problem of carrying out early development and testing activities is the absence of a hardware platform on which to run it. Other points to keep in mind are the effort and risks associated with bringing up hardware dependent software such as boot loaders. The classic approach for developing this kind of software has been to design the hardware, make a physical prototype, write the code, and then integrate the hardware and software. This methodology is nowadays too slow, and calls for an alternative to the traditional software-after-hardware design flow in order to get started with software development and testing before the hardware is ready. To shortcut these issues the SRG has developed Leon2ViP, a LEON2 virtual platform with fault-injection capabilities, which has been built around SystemC/TLM2 interfaces given previous experiences with LEON3 systems [8]. For the ICU boot and application software development, Leon2ViP provides faster edit, compile and debug cycles and, at the same time, a more controllable and observable environment for the verification activities. Furthermore, even if the hardware was already available, this virtual platform offers, not possible otherwise, non-intrusive and exhaustive debug and fault injection capabilities. The overall ICU hardware/software co-design is described in [33].

A key point of the proposed framework is that it enables the work of the design and verification teams to be decoupled. As a design principle, the Leon2ViP TLM2 code that implements system components like Instruction Set Simulators, memory modules or SpaceWire network interfaces should contain just functional code in order to emulate the behaviour of the component and represent a functional golden

model of the hardware. All fault injection code employed to emulate memory stuck-at faults must be applied in TLM2 transaction interfaces and not embedded in the model's code. This leads to the necessity of intercept TLM2 calls to the memory modules in order corrupt data read or written from/to memory or peripherals.

The interception library presented in this work is not intended to validate the Leon2ViP virtual platform itself. In fact, the library is part of the virtual platform and has been developed to help in the ICU boot software development and testing of the basic recovery mechanisms in those cases when the nominal boot sequence is not possible. It has been also used in the co-design of the SpaceWire core used for communications from/to the spacecraft.

Although the interception library has been developed specifically for the Leon2ViP virtual platform, it can used as a separate instrument in order to insert wrappers in TLM2 designs. What these wrappers are used for is up to the library user. In section 5, shows an example of how the wrapping library has been used to insert a fault injection wrapper in order to simulate stuck-at zero faults in memory access. The goal is to verify the correctness, according to the specifications of the software that runs on the virtual platform.

2.2 Paper contribution

This paper presents the results of attempting to provide a generic framework that provides dynamic binary instrumentation to TLM2 models. The approach is based on the mechanism used by C++ to implement late binding in virtual method calls. Our targeted application is the Leon2ViP virtual platform.

1. To our knowledge, this is the first library using C++ virtual table hooking as a dynamic binary instrumentation technique in order to intercept TLM2 socket primitives. The basic idea was introduced in [5] and in order to validate the approach a few code snippets specifically tailored for Microsoft VisualC++ compiler was given. This work presents a full development of the idea, intercepting all TLM2 interfaces and taking into account the compiler differences. Moreover, an implementation for the most popular compilers, VisualC++ and GCC is free released.

2. The proposed library is able to insert wrappers into the transaction path without modifying the TLM2 model source code description. This makes the technique useful for the validation of third party Intellectual Property (IP) TLM2 cores, which can be distributed as object modules. Thus, no knowledge about the source code nor methods names are needed in order to insert wrappers into the transaction communication path.

2.3 Related Work

As regards the use of virtual platforms, SystemC/TLM2 is an interoperability standard for memory-mapped bus modeling and is a key enabler for the development of virtual platforms, serving as a bridge between hardware and embedded software designers, especially for hardware-dependant and communication software development [31, 38]. Beyond the use of TLM2 for processor buses modeling, TLM2 extensions have been proposed in order to model embedded system networks. For example the work [2] proposes an extension of transaction level modeling to perform system/network design-space exploration in Networked Embedded Systems (NESs).

Using SystemC Transaction Level Modeling (TLM) it is possible to model mixed hardware/software systems in order to simulate the software behaviour in the presence of faults in the hardware. For example, works [4,35] use this methodology for the design and testing of fault tolerant systems implemented on an FPGA platform with different types of diagnostic techniques. The experimental results show the fault coverage and how Single Event Upset (SEU) occurrences cause faulty behaviours in the implemented systems. [30,40] use the same approach to verify the software of networked embedded systems long before the final hardware is available.

Coming from industrial environments, the work [24] describes the enhancement of a previous tool that allows an effective transition from the system level development phase to the software level development phase, throughout a case study based on a hybrid electric vehicle development. Another work [19] presents a system-level co-design and co-verification case study. In this work, a processor bus functional model (BFM) is used to combine native software execution with a cycle-accurate interconnect simulator and an HDL simulator.

Fault injection is mandatory in experimental dependability evaluation. Thus, the work of [25] introduces fault injection methods for Register Transfer Level (RTL) system descriptions into SystemC. Related to TLM, the work [3] presents an example of system-level fault injection in untimed functional TLM models based on FIFO channels. Related to fault tolerant TLM2 designs, the work described in [11] proposes a hardening method for inter component communication protocols.

The Assertion-Based Verification (ABV) of SystemC/TLM models is a wide field of research. Assertions capture specifications of the system being designed in an executable form. Then, they act as monitors during the simulation, detecting and reporting errors close to their source as well as establishing coverage information. Several works, such as [12, 17, 26], among others, use this approach to perform system level verification. Other works use the same approach to perform tracking/snooping of the transactions interchanged between TLM modules, for example [16, 23]. Assertions are usually written in specific languages such as Property Specification Language (PSL) and they must be translated to specific assertion code that must be placed in between the transaction's initiators and targets.

For the perspective of this work the most important issue is how the assertion code is introduced into the system model description. A common way to add additional code at specific points of the source code is by using Aspect Oriented Programming (AOP). In AOP one *aspect* is a feature linked to some parts of a program, but which is not related to the program's primary function. It is based on source-to-source translation and allows invoked methods to be wrapped with pre/post condition checkers. From this point of view *aspects* can be seen as a way of inserting wrappers into the transaction path. AspectC++ [37] is an aspect-oriented extension of C and C++ languages and is widely used to add *aspects* to SystemC descriptions [36, 39].

The previous works mainly use AOP for the verification of the hardware model described in SystemC, either RTL or TLM. The closest works to ours are [20, 21]. In these works transactions checkers are inserted in a virtual platform based on TLM2 interfaces. The goal of the checkers is to detect the wrong duration and sequence of the transactions in the TLM2 design.

As said before, the insertion library described in this work was not designed with the aim of verifying hardware TLM2 models, but for the verification and monitoring of the software running on a virtual platform built around TLM2 interfaces. However, it can also be used for other purposes as discussed in the next section.

3 TLM2 Wrapper Insertion Library Design

Talking about the instrumentation of a model description there are basically two features to consider, see Figure 1. The former is the knowledge and availability of the model source description in order to know where to instrument. The latter is the binding procedure, this is how the instrumentation code is inserted into the model.

In the case of the interception library proposed in this work, it is important to emphasize that its main goal is to provide basic services in order to intercept TLM2 transactions at runtime. This means that the modules already exist, their TLM2 sockets have been bound, SystemC has ended its elaboration phase and simulation has begun. Even more, instrumentation should be carried out without access or knowledge of the TLM2 model source code, just the TLM2 socket name of the target is known. The need to use runtime binary instrumentation on the TLM2 socket inter module bindings stems from this.

Figure 2 shows several ways of intercepting the transactions exchanged between two TLM2 modules.

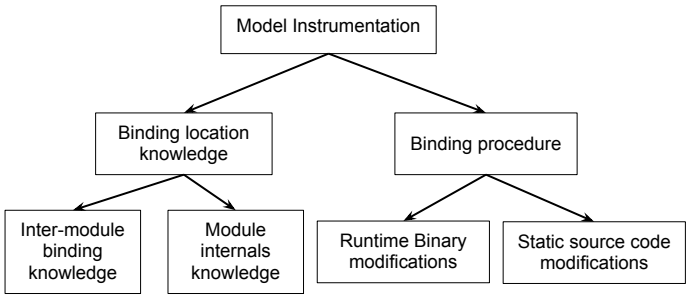


Figure 1. Instrumentation Features

The location of the inserted interception or verification code is stressed in all cases. One transaction initiator, a SPARC Instruction Set Simulator (ISS), is bound to a transaction target and uses the TLM2 non-blocking transport interface `nb_transport_fw/nb_transport_bw` to carry out transactions. These transactions are supposed to be memory read/write operations. The usual way to intercept the transaction path is the use of an interposition module placed in between the modules, (see the first case in Figure 2). In this case the initiator is bound to a module inserted into the transaction path and the interposition module is bound to the target. The second case in Figure 2 shows a C++ aspect placed in the target module in order to intercept forward non-blocking transaction calls. The `nb_transport_fw` call is intercepted by means of an `nb_transport_aspect`, so the pre-processing of the incoming transaction and post-processing of the results can be carried out. It is important to point out that in all the aforementioned approaches a source code modification is required to perform the transaction path modification. The initiator must be bound to the interceptor module or to the target. All of these bindings are done at compile time and do not change during the execution of the model. Even more, to code those bindings a profound knowledge of the TLM2 model source code is needed. For example, to insert an `aspect` around a method, the name of the method must be known. This is known as static code instrumentation.

The objective of the proposed library is to allow the insertion and removal of interposition code at runtime, as is shown in the third case in Figure 2. No modifications are made to the TLM2 modules and the original socket binding established at compile time is modified at runtime without knowledge of the modules source code. Only the name of the TLM2 socket is necessary. It is important to emphasize that the library presented in this work just provides the basic services to intercept TLM2 transactions. The specific transaction processing is built using these basic services. Once a transaction path is intercepted by means of a wrapper, it can be used in several testing scenarios such as: transaction tracking or snooping, experimental dependability evaluation through fault injection, property assertions, TLM2 protocol compliance verification, etc. As an example, [6] uses the technique being described in order to track the transactions between TLM2 components. The work [7] uses the same technique to insert a TLM2 protocol compliance checker in a non-blocking transport scenario.

The last case shown in Figure 2 is not an interception scenario. It is shown to describe the relationship of the interception scenario with other testing technologies such as *e* language (IEEE 1647). *e* is a Hardware Verification Language (HVL) mainly tailored to implement verification test benches. In this case an *e* test bench running in the initiator is bound to a TLM2 socket [1, 15]. The test bench provides the stimuli to carry out the verification of the target module.

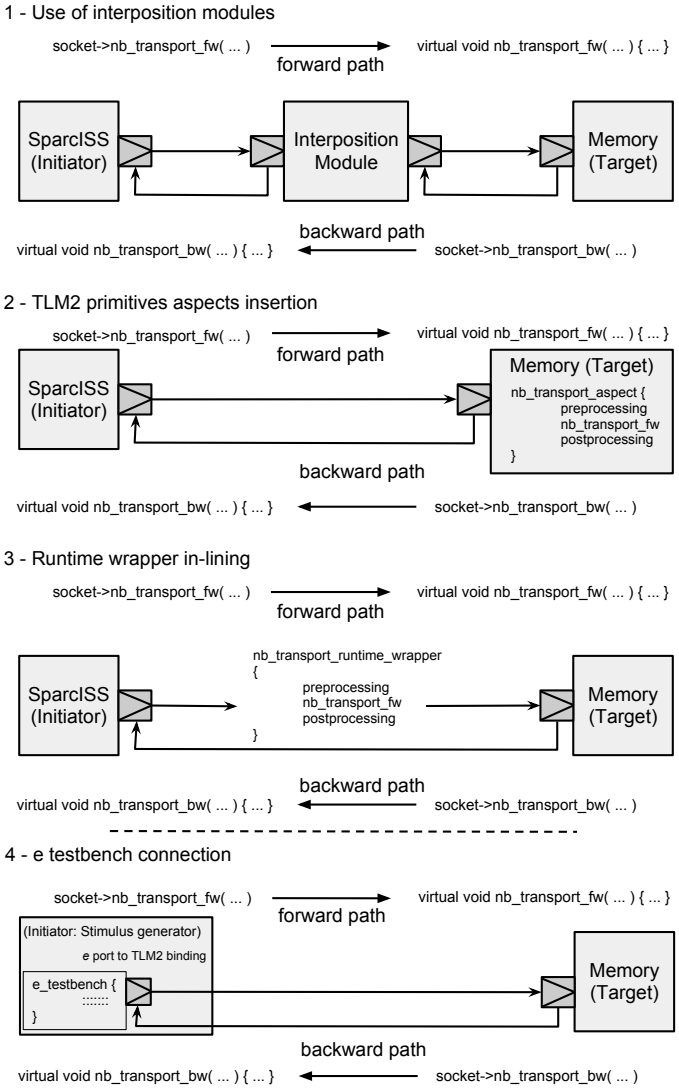


Figure 2. Interposition techniques and testbench languages

3.1 TLM2 Interfaces Definition

SystemC is a C++ object oriented framework for the description and simulation of systems. All of the system's building blocks, from basic signals to the abstract transaction level interfaces, are described using a class hierarchy, where complex classes are defined from the basic ones by means of inheritance. In a class hierarchy, it is common to find classes which define only an interface for its derived ones. No instance of a base class is actually created, only a description of an interface is given. This is done in C++ making the base class abstract, which means that at least one method is declared as pure virtual. When an abstract class is inherited, all pure virtual methods must be implemented, or the inherited class becomes abstract as well. Creating a pure virtual method allows an interface to be described without being forced to provide an implementation. The derived class must provide its own specific version of the virtual method. Several design patterns use polymorphism to invoke different functionality through a unique and standard interface. Polymorphism is a key concept in interfaces definition in C++ software designs.

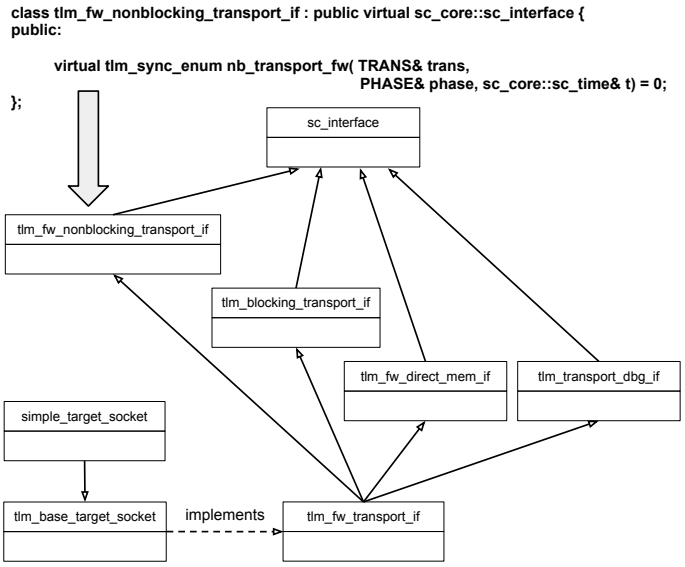


Figure 3. SystemC/TLM2 C++ forward interfaces hierarchy

The basic hierarchy of the TLM2 `tlm_fw_transport_if` interface implemented by a target socket is shown in Figure 3. This interface inherits the descriptions of four transport methods:

- `tlm_fw_nonblocking_transport_if`
- `tlm_blocking_transport_if`
- `tlm_fw_direct_mem_if`
- `tlm_transport_dbg_if`

All interfaces are abstract classes and use pure virtual methods to define the supported calls, their incoming parameters and their return values. As an example, the code of the `tlm_fw_nonblocking_transport_if` interface is also shown in Figure 3; it defines only one pure virtual method called `nb_transport_fw`. Every instance of a target socket that implements this interface must provide its own implementation of the `nb_transport_fw` method.

3.2 C++ Polymorphism implementation

Polymorphism is, with inheritance, one of the essential features of an object-oriented programming language like C++. It provides separation of interface definition from the particular implementation of that interface, decoupling "what" from "how". The virtual methods allow one type to express its distinction from other similar type, as long as they are both derived from the same base type. The distinction is expressed through a different implementation of the methods' behaviour. These methods must be called through base class pointers.

Connecting a function call to a function body is called *binding*. When binding is performed at the time of compilation, it is called early binding. On the other hand, late binding means the binding occurs at runtime, depending on the class of the object. Late binding is also called dynamic binding or runtime binding. When a language implements late binding, there must be some mechanism to determine the class of the object at runtime and call the appropriate method. In the case of a compiled language, the compiler still does not know the actual object class, but it inserts code that finds out how the invocation has to be resolved and finally calls the right method. Late binding only occurs with virtual methods, and only when the call is made through base class pointers.

Figure 4 describes the "big picture" of a virtual call. Each time a class containing virtual methods is created or derived from a class that contains virtual methods, the compiler creates a unique virtual method address table (VTABLE) for that class. In this table it places the addresses of all the methods that have been declared virtual in this class or in the base class. It is by using this table that the addresses of the invoked methods are obtained at runtime. Note that virtual tables are class specific and that there is only one virtual table for each class regardless of the number of object instances.

It is possible to modify the VTABLE and insert the address of a wrapper method to carry out the necessary processing of the transaction parameters and monitor the behaviour of the system. What is more, the VTABLE can be duplicated so that the change only affects a particular class instance and not all of them. This is a great improvement on AOP programming, for it applies to the class definition regardless of the number of instances of the class. When wrapping is no longer needed, the VTABLE can be restored to its original value. It is important to note that the particular layout of the VTABLE is not explicitly defined in the C++ Application Binary Interface (ABI) so some particularities are C++ compiler dependant. However, all C++ compilers use similar approaches. Common C++ compilers have undocumented compilation switches that export the class layout. For Microsoft VisualC++, `-d1reportAllClassLayout` can be used. For GCC's g++ compiler, the switch is `-fdump-class-hierarchy`. The following tools and works are useful to find out the internal structure of C++ programs [14, 22, 34].

3.3 Binary Layout and Compiler Dependencies

When a call to a virtual function is made through a base class pointer (i.e. a late binding call), the compiler quietly inserts code to fetch the VTABLE pointer (VPTR) and look up the requested method address in the VTABLE, thus calling it and causing late binding to take place. All of this VTABLE setting up for each class, initializing the VPTR and inserting the code for the virtual function call, happens automatically. Figure 5 shows the binary layout of a `tlm_fw_transport_if` object and how a `b_transport` virtual method invocation using two different object pointers, is resolved from the point of view of Microsoft VC++ and GCC 4.1 compilers. This situation is more complex since two key object-oriented features, such as polymorphic calls and multiple inheritance, are used at the same time. For

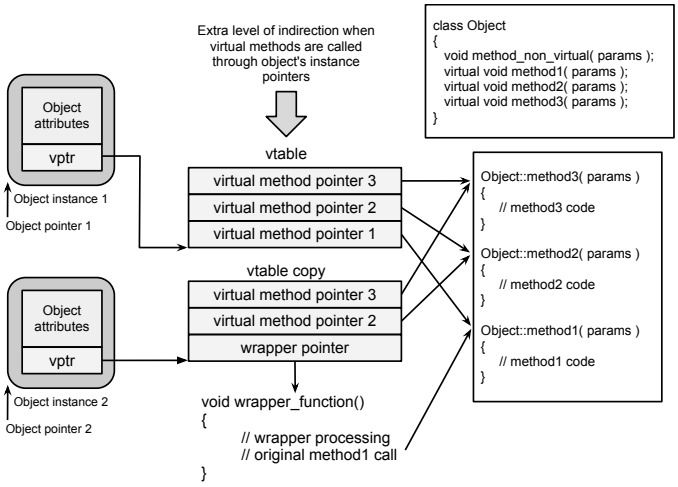


Figure 4. C++ Virtual Table Basic Structure

TLM2 interfaces the arrangement of both compilers is as follows:

- Microsoft VisualC++ (VC) distributes the information of each inherited class by placing them one next to another. If an inherited class has virtual methods, a pointer to its particular VTABLE is included. Since each interface has only one virtual method, there are four VTABLEs with only one entry pointing to the particular implementation of each method. When a call is made through a derived class pointer, `tlm_fw_transport_if *p_fw` in Figure 5, the pointer is adjusted to point to its specific VPTR and class data. This pointer modification is known as *pointer fix-up*. Finally, since each interface has only one method, the first entry of the current VTABLE (i.e. entry 0) is used to call the right method.

- The GCC 4.1 compiler places information of all classes at the beginning of the object. Depending on the kind of pointer used to call the method, different sets of VTABLEs are used. If a call is issued through the object's pointer, again `tlm_fw_transport_if *p_fw` in Figure 5, the derived class' VTABLE is used. On the other hand, if the call is made by means of the specific class pointer, for example `tlm_fw_transport_if *p_b`, the destination address is taken from its particular VTABLE. The latter case leads to the need for a pointer fix-up before the method code begins in order to allow access to the class' data. This pointer adjustment code is known as `thunk code`.

In short, VC makes a pointer adjustment before VTABLE lookup while GCC does it afterwards.

3.4 Wrapper Library Design

As the TLM2 interface methods are all "virtual", every call is made through the VTABLE of the object instance. Therefore it is possible to modify the VTABLE to insert some kind of function wrapper without modifying the original source code as described in Figure 4. The class diagram shown in Figure 6 details the relationship between all the elements that model the wrapper infrastructure. For simplicity, not all the attributes and methods are shown. The base class of the wrapper library is `TLM2_Wrapper`. This class

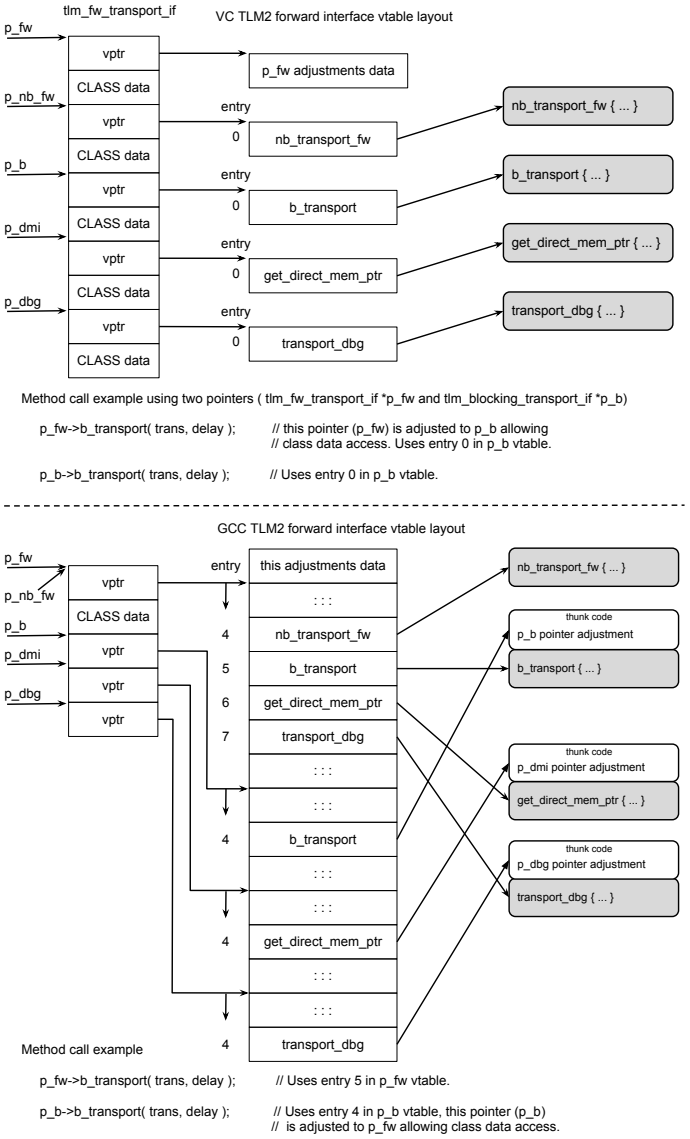


Figure 5. Microsoft VC vs GCC TLM2 Forward Interface Binary Layout

provides the interface to handle all TLM2 interfaces. Since all of them receive the same processing, only the `b_transport` interface is explained on the understanding that the others work in the same way.

Here it is important to understand the "is a" and "has a" relationship between classes. `TLM2_Wrapper` is the base class of the hierarchy and provides a basic pass-through wrapping service. This basic behaviour can be particularized by the use of inheritance, redefining just one method of the base class. For example, the `Fault_Injection_Wrapper` in Figure 6, provides its own version of the `b_transport_processing` method. This class "is a" specialized `TLM2_Wrapper`.

On the other hand, the `TLM2_Wrapper` uses the low level services provided by the `WIL` class to modify and restore the VTABLE of the wrapped interface. This is a "has a" relationship. "WIL" stands for Wrapper Interception Library and provides the methods to get, set and reset the specific VTABLE entries of each TLM2 interface, taking into consideration the compiler binary dependencies. The declared interface for the `b_transport` handling is made up of the following methods:

- `b_transport_wrapper`: The start address of this method is inserted into the corresponding VTABLE entry of the intercepted target socket. It just redirects the call to `b_transport_processing`.
- `b_transport_processing` (virtual): This method implements the desired behaviour of the wrapper. The default implementation in the base class just calls `b_transport_original_path` resulting in a pass-through mode. As it is declared virtual, this method is intended to be redefined in the derived class for specialized behaviour.
- `b_transport_original_path`: This method is used to call the original entry of the modified VTABLE in order to maintain the overall transaction path.

The `WIL` class has four methods:

- `wil_get_fw_interface`: For internal use. This method gets the TLM2 forward interface pointer given the hierarchical SystemC name of the socket.
- `wil_get_bw_interface`: For internal use. This method gets the TLM2 backward interface pointer given the hierarchical SystemC name of the socket.
- `wil_dup_vtable`: Duplicates the VTABLE of an object, given its forward and backward TLM2 interface pointers.
- `wil_reset_vtable`: Restores the original VTABLE of an object, given its forward and backward TLM2 interface pointers.

The specific handling methods for each interface are inherited from the `WIL_b_transport`, `WIL_nb_transport_fw`, `WIL_debug_transport` and `WIL_dmi` classes. The handling methods for `WIL_b_transport` are:

- `wil_get_b_transport`: Gets the associated VTABLE entry value, given the hierarchical tical SystemC name of the socket.
- `wil_set_b_transport`: Sets a specific VTABLE entry with a new value. This new value is supposed to be a wrapper method address.
- `wil_reset_b_transport`: Restores the original value of a VTABLE entry.
- `wil_get_mapped_b_transport_entry`: For internal use. Gets the original entry mapped onto the actual wrapper. This allows the original values to be restored when it is desired.

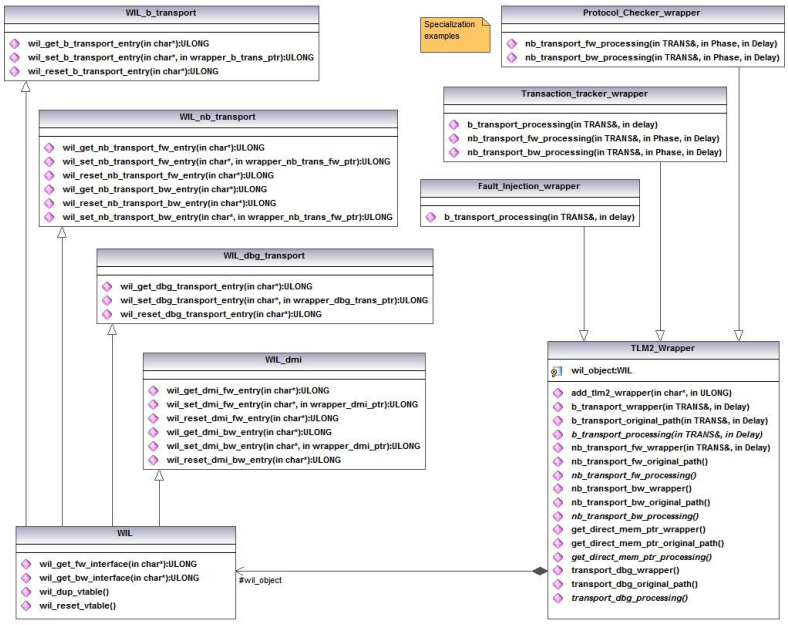


Figure 6. TLM2 Wrapper Class Diagram

Figure 7 shows the method's call sequence in a TLM2 `b_transport` intercepted environment. The main actor is an initiator socket bound to a target socket. The first sequence shows the nominal sequence when the initiator issues a `b_transport` call. As expected, the corresponding `b_transport` method is executed in the target. Next, in order to intercept these kinds of calls and track the information interchanged between both modules, a `Fault_Injection_Wrapper` is created by means of the C++ `new` operator. After the creation of the wrapper, the `add_tlm2_wrapper` method is invoked to insert the wrapper into the transaction path. To do this, the hierarchical SystemC name of the initiator socket is passed as a parameter. The other parameter is a constant value indicating the specific interface to be intercepted, `B_TRANSPORT` in this example. Once the wrapper is inserted, the same call issued by the initiator in the first sequence, is now redirected to the `b_transport_wrapper` method. As said, the default behaviour of this method is to call `b_transport_processing` but since it is declared virtual and redefined in the derived class, the `Fault_Injection_Wrapper` version of this method is called. `b_transport_processing` can now carry out a pre-processing of the incoming transaction parameters before calling the `b_transport_original_path` method that maintains the original transaction path. After the transaction is processed by the target socket, the call flow returns again to `b_transport_processing` allowing some kind of post-processing of the returned values. Finally, the return of this method leads to the completion of the original `b_transport` transaction call.

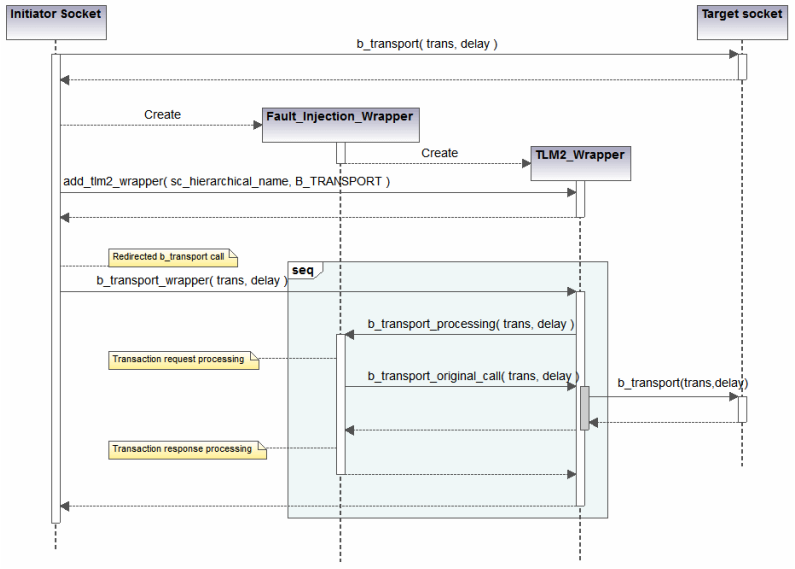


Figure 7. TLM2 Wrapper Sequence Diagram

3.5 Usage example

The following code snippets show an example of how to use the library. First, the declarations of an initiator and a target, both with a TLM2 socket attribute, are:

```
struct Initiator: sc_module
{
tlm_utils::simple_initiator_socket<Initiator> socket;
::::

struct Target: sc_module
{
tlm_utils::simple_target_socket<Target> socket;
::::
```

Given the previous declarations, the object instances are obtained through the `new` operator. Finally, both sockets are bound through the bind method of the initiator socket. From this moment, every TLM2 method call through the initiator socket brings about the execution of the corresponding response method in the target module.

```
p_initiator = new Initiator("Initiator");
p_target    = new Target   ("Target");

p_initiator->socket.bind( p_target->socket );
::::
```

If it is necessary to intercept the call to the `b_transport` method of the target module, it is enough to define a `tlm2_wrapper` class with the particular processing method. In the example just `b_transport_processing` method

is defined. This method can carry out some pre and/or post processing of the incoming call parameters, as well as calling the original target method as is shown below.

```
void tlm2_wrapper::b_transport_processing( transaction_type& trans,
                                           sc_core::sc_time& delay )
{
//    pre_processing( trans, delay );
cout << "---B_TRANSPORT Pre_wrapper---" << endl;

b_transport_original_path_call( trans, delay );

//    post_processing( trans, delay );
  cout << "---B_TRANSPORT Post_wrapper---" << endl;
}
```

Finally, the wrapper class is instantiated through the C++ `new` operator. The constructor receives the hierarchical name of the initiator module as a parameter as defined by SystemC. By means of the `add_tlm2_wrapper` method, the wrapper is inserted into the transaction path. In this example only the `B_TRANSPORT` interface is intercepted.

```
tlm2_wrapper  *p_wrapper = new tlm2_wrapper( "top.Initiator.socket_0" );
::::
p_wrapper->add_tlm2_wrapper( B_TRANSPORT );     // Wrapper inlining for
                                                // B_TRANSPORT interface
::::
top.p_initiator->socket->b_transport( *trans, delay ); // Call does't change
::::
p_wrapper->remove_tlm2_wrapper( B_TRANSPORT );  // Wrapper removing
```

4 Performance and Usability Issues

The interception library has been coded in a C++ object oriented style. This makes the specialization of the wrappers very easy through inheritance. The library has less than 2.000 lines of code as shown in Table 1 along with the amount of code that is compiler dependent.

Total lines	VisualC++ Specific Code Lines	GCC Specific Code Lines
1895	109	44

Table 1. Interception Library Compiler Dependant Code

The source code of TLM2 Wrapper Library V1.0 can be found at `https://dl.dropbox.com/u/19987939/TLM2Wrapper.zip`. The usage example distributed in the main module defines just two TLM2 modules and binds them in the usual way. After that a pass-through TLM2 wrapper is instantiated and the wrapper is inlined in the original transaction path. After invoking all TLM2 primitives, wrapper is removed and main program ends. Table 2 summarizes the code coverage of the different modules. Data has been obtained through GCOV, the GNU source code coverage analysis tool. As shown in Figure 6 `wil_b_transport` and `wil_dbg_transport` have three methods plus class constructor, `wil_nb_transport` and `wil_dmi` have six methods plus class constructor and `wil` class inherits the twenty-two methods of the previous classes plus its own class constructor. As seen in Table 2 all basic operations for inserting/removing wrappers in TLM2 primitives are all exercised.

Table 3 summarizes the results of a series of performance tests carried out to estimate the overhead associated with the insertion of the wrapper into the transaction path using different kinds of techniques. Some usability considerations are also included. All tests were carried out using a 1,66 GHz generic laptop with 1 Gbyte RAM

Module	Lines Executed	Branches	Calls
main	77.9% of 86	80.7% of 52	56.2 of 105
tlm2_wrapper	73.4% of 263	69.7% of 172	60.56% of 142
wil	95.52 of 67	100% of 50	100% of 23
wil_b_transport	89.5% of 19	100% of 12	100% of 4
wil_nb_transport	67.6% of 37	100% of 20	100% of 7
wil_dmi	67.6% of 37	100% of 20	100% of 7
wil_dbg_transport	89.5% of 19	100% of 12	100% of 4

Table 2. Interception Libary Distribution Example Coverage

Memory using Microsoft Windows 7 Professional Service Pack 1. The inserted wrapper implements a pass-through functionality and just calls the target method. No modifications to the input parameters or return values are made. For each initiator socket transaction, the processing time is measured using the time-stamp counter RDTSC [32], present in Intel processors. A series of 1.000 calls to `b_transport` target method were made using the following two configurations:

- Use of an interposition module between initiator and target.
- Use of virtual table hooking to interpose a method wrapper in the transaction path using the C++ library presented in this paper.

Insertion technique	Single call time overhead	TOP model source code modification	Insertion time	Removing time
Interposition module	30%	Yes	Compile time	Not allowed
VTABLE hooking	20%	No	Run time	Run time

Table 3. Performance and Usability Issues

Given a TLM2 model, the introduction of any interposition code always introduces some time penalties. Even more if extensibility is design concern. The results show that virtual table hooking used by the interception library introduces a time overhead close to but lower than that obtained with an interposition module. Other important point to consider is that with this technique, only the specific desired call is intercepted while the other calls of the TLM2 interface run in the usual way. Using interposition implies that all methods of the socket interface are intercepted and thus delayed. In the case of the virtual table hooking wrapper, it can be inserted and removed at runtime. Other improvement of the virtual hooking method is that it can be applied to a single object instance instead of a class. This concern is exploited in the library usage example described in section 5.

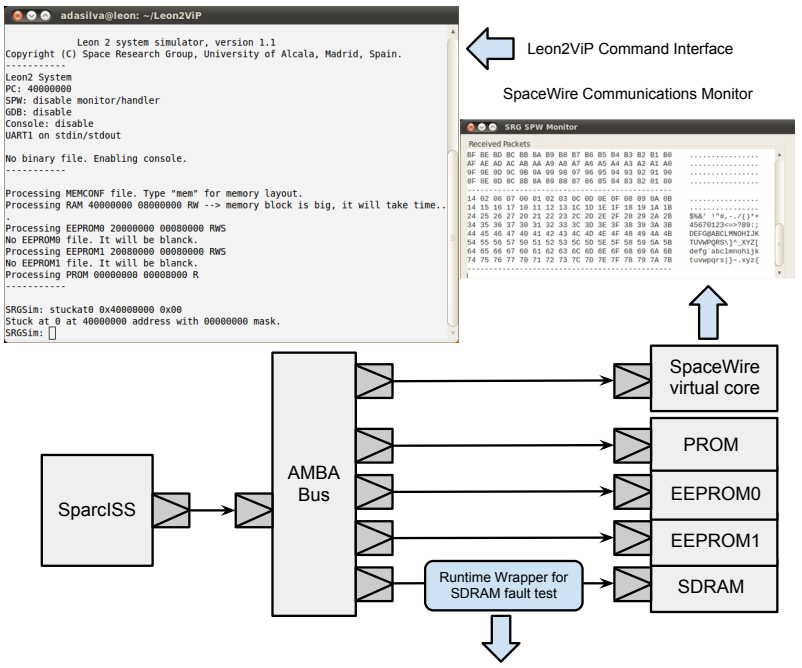
5 Inline wrapper Library Fault Injection Use Case

This section presents a real testing scenario of the ICU's boot software using the Leon2ViP virtual platform and the TLM2 wrapper insertion library. Since the internal architecture of the Leon2ViP virtual platform is not the main focus of this paper, Figure 8 just shows only the main components, stressing the location of the fault injection wrappers:

- LEON2 ISS: SPARC V8 untimed Instruction Set Simulator with blocking TLM2 transaction interfaces.
- Memory: PROM, EEPROM and SDRAM blocks. The memory layout is highly configurable through an external configuration file and the current contents can be read from an external ordinary binary file generated by the compiler toolchain.
- Bus: This module interconnects all the TLM2 components of the virtual platform.

- SpaceWire: Virtual SpaceWire IP core for spacecraft communications. This interface can be mapped onto a SpaceWire monitor or an external SpaceWire hardware based on a Star-Dundee USB SpaceWire brick [10].

For controlling the software execution, Leon2ViP offers a command-line interface with which the user can issue commands for loading memory binaries, for inspecting the internal state of the processor and the memory blocks and for defining breakpoints/watchpoints, among many others. SDRAM and PROM areas are usually filled with external program binary files using a `load` command. Leon2ViP also emulates EEPROM areas, the memory contents of these areas are filled at the system start-up with the contents of a file which has the same name as the memory block. The file should have the same size as the memory block and must contain its binary image. When the execution ends, the contents of the memory blocks are written back to the files. This emulates the behaviour of an EEPROM when data updates are carried out on it using the Software Data Protection (SDP) schema implemented in commercial EEPROM modules.




```
void tlm2_parser_wrapper::b_transport_processing( transaction_type& trans, sc_core::sc_time& delay )
{
   // cout << "---B_TRANSPORT Wrapping---" << endl;

   // if WRITE command and address match, apply MASK on data that is going to be written

   // Call target original b_transport  on SDRAM
   b_transport_original_path_call( trans, delay );

   // if READ command and address match, apply MASK on data read
}
```

Figure 8. SoLO bootloader fault injection framework

One of the strictest requirements of the flight software is that the application binaries should be updatable in flight. This characteristic allows the system to recover from application bugs discovered after launch itself. There are two versions of the ICU application software. EEPROM0 module contains the *Baseline* version and EEPROM1 module contains an *Updatable* version. In accordance with the ICUSW robustness requirements [28], the boot software takes control of the ICU automatically after a reset or a power-on and is in charge of deploying the main application software. The boot software verifies the integrity of the two versions of the application software stored in the EEPROM modules, as well as the integrity of the memory on which they are deployed. Depending on the result of this checking, it will deploy one of the two versions of the application software. If both application software images are damaged, the boot software will wait and remain active until the images have been repaired by means of service telecommands, sent from Earth, that allow memory patch, dump, and check operations. The SDRAM will be tested in order to find possible failures that may condition the system's stack allocation and application binary deployment resulting from stuck-at faults. The stack allocation takes into account that only 32 Kibibytes of not damaged SDRAM are required for running the boot software. Again, when neither of the software application versions can be deployed, the ICU boot software must wait for spacecraft telecommands that will load a new version of the application software that avoids the damaged SDRAM areas.

In Figure 8 the user issues a `stuckat0` command at the 0x40000000 address. This means that every read/write transaction on the selected address must be tied to a logical zero. To carry out this memory access corruption a wrapper is inserted into the SDRAM memory path. As long as the wrapper is inserted the fault remains active. The fault injection wrapper developed using the interception library just redefine the `b_transport_processing` method in order to apply the mask to the data read/written in the transaction and is inserted just into the transaction path of the memory module under fault simulation. For example, in order to test the boot behaviour when SDRAM has stuck-at faults the fault injection wrapper is inserted in the SDRAM path. This leaves the PROM and EEPROM transaction paths unchanged, thus fetch operations on PROM boot code and read operations on EEPROM modules are not delayed. A complete description of the first results on testing ICU boot can be found in [9].

6 Conclusions and future work

In this article, we have presented a library for runtime wrapper insertion in TLM2 models. The results presented here are encouraging, and show that it is possible to insert transaction wrappers in an easy way with a minimal time overhead and with a great improvement on previous approaches such as interception modules. The proposed technique has been successfully applied on a real early software development and robustness verification scenario. In addition, the technique adopted here is applicable in third-party software component validation, without having access to the component source code. It has the disadvantage of using some not-well-documented and compiler-dependant features. The library design has been made by thinking about its usability and easy customization. This feature greatly reduces deployment times for real-world testing scenarios. The runtime nature of the wrapper inlining has an improved effect overall system simulation since wrappers can be installed on the fly if needed or when some special runtime situation in the system is reached. In the same way, the wrappers can be removed when they are no longer needed. In addition, since the insertion/removal of wrappers is time-consuming and error-prone, automating this task also ensures that the testing process does not compromise the correctness of the final system. A C style version of the library is under development in order to reduce the time penalty introduced in the current version for a single call.

Acknowledgments

This work has been supported by Spanish Ministerio de Economía y Competitividad under the grants AYA2011-29727-C02-02 and AYA2012-39810-C02-02.

References

1. Brian Bailey, Felice Balarin, Michael McNamara, Guy Mosenson, Michael Stellfox, and Yosinori Watanabe. *TLM-Driven Design and Verification Methodology.* Lulu Enterprises Inc., 2010.

2. Nicola Bombieri, Franco Fummi, and Davide Quaglia. System/network design-space exploration based on TLM for networked embedded systems. *ACM Trans. Embed. Comput. Syst.*, 9(4):37:1–37:32, April 2010.
3. K. J. Chang and Y. Y. Chen. System-level fault injection in SystemC design platform. In *Proc. 8th Int. Symposium on Advanced Intelligent Systems*, pages 354–359, 2007.
4. Sergio Cuenca-Asensi, Antonio Martínez-Álvarez, Felipe Restrepo-Calle, Francisco R. Palomo, Hipolito Guzmán-Miranda, and Miguel A. Aguirre. Soft core based embedded systems in critical aerospace applications. *Journal of Systems Architecture*, 57(10):886 – 895, 2011.
5. Antonio da Silva and Sebastián Sánchez. On the use of Dynamic Binary Instrumentation to perform faults injection in transaction level models. In *Dependability of Computer Systems, 2009. DepCos-RELCOMEX '09. Fourth International Conference on*, pages 237 –244, 30 2009-july 2 2009.
6. Antonio da Silva and Sebastián Sánchez. Transactions sequence tracking by means of Dynamic Binary Instrumentation of TLM models. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on*, pages 723 –728, aug. 2009.
7. Antonio da Silva and Sebastián Sánchez. A grammar based testing framework for TLM2.0 protocol compliance verification. In *Technical Approach to Dependability. Proceedings of RELCOMEX 2010: Fifth International Conference on Dependability of Computer Systems DepCoS*, Monographs of System Dependability, pages 121–132, Wroclaw, Poland, 2010. Oficyna Wydawnicza Politechniki Wroclawskiej.
8. Antonio da Silva and Sebastián Sánchez. LEON3 vip: A virtual platform with fault injection capabilities. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pages 813 –816, sept. 2010.
9. Antonio da Silva, Sebastián Sánchez, Óscar R. Polo, and Pablo Parra. Injecting faults to succeed. verification of the boot software on-board Solar Orbiter's Energetic Particle Detector. *Acta Astronautica*, 95(0):198 – 209, 2014.
10. STAR Dundee. Star dundee brick, 2013.
11. Ali Ebnenasir, Reza Hajisheykhi, and Sandeep S. Kulkarni. Facilitating the design of fault tolerance in transaction level SystemC programs. *Theoretical Computer Science*, (0):–, 2012.
12. Wolfgang Ecker, Volkan Esen, Thomas Steininger, Michael Velten, and Michael Hull. Interactive presentation: Implementation of a transaction level assertion framework in SystemC. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '07, pages 894–899, San Jose, CA, USA, 2007. EDA Consortium.
13. ESA. Solar orbiter exploring the sun-heliosphere definition study report. Technical Report I-CA2301, European Space Agency, 2011.
14. A. Fokin, K. Troshina, and A. Chernov. Reconstruction of class hierarchies for decompilation of C++ programs. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 240 –243, march 2010.
15. Hannes Froehlich. *Interface Additions to the e Language for Effective Communication with SystemC TLM 2.0 Models*. Cadence, 2012.
16. A.A. Ghofrani, S. Abolma'ali, Z.N. Haghi, and Z. Navabi. A TLM2.0 assertion library with centralized monitoring approach. In *Design Test Symposium (EWDTS), 2010 East-West*, pages 402 –406, sept. 2010.
17. Ali Habibi and Sofiene Tahar. Design and verification of SystemC transaction-level models. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(1):57 –68, jan. 2006.
18. Thomas A. Henzinger. Two challenges in embedded systems design: predictability and robustness. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366(1881):3727–3736, 2008.
19. Sungpack Hong, Tayo Oguntebi, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. A case of system-level hardware/software co-design and co-verification of a commodity multi-processor system with custom hardware. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '12, pages 513–520, New York, NY, USA, 2012. ACM.

20. M. Kallel, Y. Lahbib, R. Tourki, and A. Baganne. Aspect-based ABV for SystemC transaction level models. In *Microelectronics (ICM), 2009 International Conference on*, pages 304 –307, dec. 2009.
21. M. Kallel, Y. Lahbib, R. Tourki, and A. Baganne. Verification of SystemC transaction level models using an aspect-oriented and generic approach. In *Design and Technology of Integrated Systems in Nanoscale Era (DTIS), 2010 5th International Conference on*, pages 1 –6, march 2010.
22. Nicholas A. Kraft, Brian A. Malloy, and James F. Power. A tool chain for reverse engineering C++ applications. *Sci. Comput. Program.*, 69(1-3):3–13, December 2007.
23. Pierre Laurence and Luca Ferro. A tractable and fast method for monitoring SystemC TLM specifications. *IEEE Trans. Comput.*, 57(10):1346–1356, October 2008.
24. Roland Mader, Gerhard Griessnig, Eric Armengaud, Andrea Leitner, Christian Kreiner, Quentin Bourrouilh, Christian Steger, and Reinhold Weiss. A bridge from system to software development for safety-critical automotive embedded systems. *2012 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, 0:75–79, 2012.
25. S. Misera, H. T. Vierhaus, and A. Sieber. Simulated fault injections and their acceleration in SystemC. *Microprocess. Microsyst.*, 32(5-6):270–278, 2008.
26. B. Niemann and Ch. Haubelt. Assertion-based verification of transaction level models. In *Proceedings of In ITG/GI/GMM Workshop*, volume 9, pages 232–236, 2006.
27. OSCI. Open SystemC Initiative, systemc, 2013.
28. Ó. R. Polo and P. Parra. Solo epd flight software requirements(2012). Technical report, Space Research Group, U. Alcalá), 2012.
29. Luis Rabelo, Serge Sala-Diakanda, John Pastrana, Mario Marin, Sayli Bhide, Oloruntomi Joledo, and Jorge Bardina. Simulation modeling of space missions using the high level architecture. *Modelling and Simulation in Engineering*, 2013, January 2013.
30. Lu Weiyunand Martin Radetzki. Concurrent and comparative fault simulation in SystemC and its application in robustness evaluation. *Microprocessors and Microsystems*, 37(2):115 – 128, 2013.
31. Famantanantsoa Randimbivololona, Abderrahmane Brahmi, Philippe Le Meur, Thomas Marie, and Romain Beseme. Final integration test of avionic software in full virtual platform. In *Embedded Real Time Software and Systems (ERTS2 2014), Toulouse, France, Feb 2014*, 2014.
32. RDTSC. Using the rdtsc instruction for performance monitoring, 2009.
33. Sebastián Sánchez, Manuel Prieto, Óscar R. Polo, Pablo Parra, Antonio Da Silva, Óscar Gutiérrez, Ronald Castillo, Javier Fernández, and Javier Rodríguez-Pacheco. HW/SW co-design of the Instrument Control Unit for the Energetic Particle Detector on-board Solar Orbiter. *Advances in Space Research*, 2013.
34. I. Skochinsky. Practical C++ decompilation, recon 2011, 2011.
35. Martin Straka, Jan Kastil, Zdenek Kotasek, and Lucas Miculka. Fault tolerant system design and SEU injection based testing. *Microprocessors and Microsystems*, 37(2):155 – 173, 2013.
36. Deian Tabakov and Moshe Y. Vardi. Automatic aspectization of SystemC. In *Proceedings of the 2012 workshop on Modularity in Systems Software*, MISS '12, pages 9–14, New York, NY, USA, 2012.
37. Reinhard Tartler, Daniel Lohmann, Fabian Scheler, and Olaf Spinczyk. AspectC++: An integrated approach for static and dynamic adaptation of system software. *Know.-Based Syst.*, 23(7):704–720, October 2010.
38. Wang Yang, Wang Lifeng, and Zheng Zewei. Application of virtual prototype technology to simulation test for airborne software system. In *Advances in Electronic Engineering, Communication and Management Vol.2*, volume 140 of *Lecture Notes in Electrical Engineering*, pages 653–658. Springer Berlin Heidelberg, 2012.
39. Endoh Yusuke. ASystemC: an AOP extension for hardware description language. In *Proceedings of the tenth international conference on Aspect-oriented software development companion*, AOSD '11, pages 19–28, New York, NY, USA, 2011. ACM.
40. Claas Ziemke, Thoshinori Kuwahara, and Ivan Kossev. An integrated development framework for rapid development of platform-independent and reusable satellite on-board software. *Acta Astronautica*, 69(78):583 – 594, 2011.

A.5. Injecting faults to succeed. Verification of the boot software on-board Solar Orbiter's energetic particle detector

Autor

Antonio da Silva, Sebastián Sánchez, Óscar R. Polo y Pablo Parra

Lugar

Acta Astronautica, ISSN: 0094-5765, 2014, Volumen: 95, Página: 198-209

Tipo

Journal paper

Ref

SCImago 2013: Category Aerospace Engineering Q1
JCR 2013: Q2

Resumen

Se dice que incluso el viaje más largo comienza con un primer paso. Esto es también cierto para el software de aplicación. Durante el arranque, los ordenadores ejecutan un conjunto inicial de instrucciones que típicamente realizan un test de la memoria y cargan el entorno de ejecución final. Este artículo describe la verificación de los requisitos contra SEEs del software de arranque de la Unidad de Control del Instrumento (ICU) del Detector de Partículas de alta Energía (EPD) a bordo de Solar Orbiter. Puesto que en esta fase del arranque software no hay activos servicios de ningún tipo, es difícil poder llevar a cabo una verificación del software en hardware real. Para atajar este problema el Grupo de Investigación del Espacio de la Universidad de Alcalá (SRG) ha desarrollado una plataforma virtual para LEON2 (Leon2ViP) usando SystemC con capacidad de inyección de fallos. De esta forma es posible ejecutar exactamente el mismo binario como si fuera en el hardware real pero en un entorno más controlado y determinista, permitiendo una verificación más estricta de los requisitos. El uso de "Leon2ViP" ha significado una mejora importante en tiempo y coste en el desarrollo y verificación del software de arranque de la ICU.

Injecting Faults to Succeed. Verification of the Boot Software On-Board Solar Orbiter's Energetic Particle Detector[☆]

Antonio da Silva*, Sebastián Sánchez, Óscar R. Polo, Pablo Parra

Space Research Group, University of Alcala, Alcalá de Henares, Madrid, Spain

Abstract

It is said that even the longest journey begins with the first step. This is also true for application software. When the power is switched on, computer systems execute an initial set of operations that usually perform memory tests and load the final runtime environment. This paper describes the Single Event Effects (SEEs) requirements verification of the boot software that will run in the Instrument Control Unit (ICU) of the Energetic Particle Detector (EPD) on-board Solar Orbiter. Since in the booting stage there are no software services at all, it is difficult to achieve a complete software verification on real hardware. To shortcut this issue the Space Research Group (SRG) of the University of Alcalá has developed a LEON2 Virtual Platform (Leon2ViP) based on SystemC with fault injection capabilities. This way it is possible to run the exact same target binary software as if were run on the physical system, but in a controlled and deterministic environment, thus allowing a stricter requirements verification. The use of Leon2ViP has meant a significant improvement, in both time and cost, in the development and verification processes of the ICU's boot software.

Keywords: Fault injection, Verification and validation, Bootstrap testing, Single Event Effect, Virtual platforms

[☆]This work has been supported by the MINECO under the grant AYA2011-29727-C02-02.

*Corresponding author

Email address: adasilva@srg.aut.uah.es (Antonio da Silva)

1. Introduction

Because of the tough robustness requirements in space software development, it is imperative to carry out verification tasks at a very early development stage to ensure that the implemented exception mechanisms work properly. This also helps to evaluate the possible risks, revealing how the system behaves in the presence of faults. These fault tolerance requirements call for full simulation environments, in which virtual platforms allow software to be developed and tested with a high degree of accuracy at a very early hardware development stage. This is fundamental in evaluating the fault detection and recovery mechanisms implemented in the software design. The verification of software fault tolerance mechanisms implemented in critical systems to recover the system from exceptional situations can be difficult. This is because such situations must be systematically and artificially brought about during the verification phase. Fault tolerance mechanisms are often verified by means of experimental techniques such as fault injection, which comprises a variety of techniques for introducing faults into a system and modifying its behaviour to facilitate the reproduction of hidden or unforeseen problems in order to:

- verify exception handling and recovery mechanisms: in classic software testing methodologies, particular exception or error handling procedures, if any, are rarely triggered and even less tested [1];
- provide an experimental assessment of the risks [2]: the system's behaviour in the presence of faults can be used as a way to quantify potential risks of the system and to allow the prediction of worst-case scenarios.

Virtual platforms are executable models of complete systems that provide software developers with working frameworks a long time before the real hardware is available. Virtual platforms enable the concurrent development of System-on-Chip (SoC) hardware and software, significantly shortening their integration times. For embedded software development and verification some of the advantages of using virtual platforms are to:

- run the same target software binary as if on the physical system, but in a controlled and deterministic environment;
- reduce the dependencies of the software and system tasks on hardware availability;
- provide debugging and fault injection capabilities which are unattainable otherwise;
- offer the capability to connect physical devices through standard communication interfaces, thus allowing a virtual hardware-in-the-loop testing approach.

Verification is a major process in the development of aircraft and spacecraft software. Its purpose is to detect and report errors that may have been introduced during the development process. Verification is typically a combination of reviews, analyses and tests with the aim of assessing, with a high degree of confidence, that errors that could lead to unacceptable failure conditions have been removed.

Embedded software testing has been mainly carried out on dedicated hardware resources. The limitations incurred while using these dedicated resources have been known for a while: cost, availability, the use of intrusive testing techniques and the lack of debugging capabilities, observability and controllability. These limitations are noticeably more acute when dealing with the testing of fault tolerance-related properties. These tests require specific hardware and software setups, which are not always technically achievable, nor practically affordable in a non-intrusive manner. The use of a fully virtual platform is an alternative approach able to yield effective solutions to these limitations. From an embedded software perspective, the use of virtual platforms allows the development and verification processes to be started earlier in the design flow so as to detect and correct errors that would otherwise propagate to the final implementation stages. Moreover, it is easier to access and modify the internal state of the virtual prototypes, so that a comprehensive fault injection campaign and

fault tolerance assessment can be carried out.

The remainder of the paper is organized as follows: relevant related works are detailed in the next subsection. Section 2 describes the mission's characteristics. Section 3 the embedded software development and testing challenges. Section 4 describes the adopted solution based on the use of an ad-hoc virtual platform development. Section 5 describes the experimental setup used to verify several robustness software requirements along with the results. Section 6 contains the conclusions.

1.1. Related work

The use of virtual platforms gives developers far more visibility and control over system design by the very nature of its virtuality. Any state is within reach and any condition can be triggered. Therefore, virtual platforms have become widely used in design space exploration and early software development in avionics and space software environments, before the hardware becomes available [4, 5]. There are several approaches, ranging from symbolic execution to binary compatible instruction-set simulators. Current research focuses on experimental techniques and tools that allow software robustness verification through fault injection.

Several works deal with model-based verification and symbolic execution. Symbolic execution has been used for a wide variety of software testing and maintenance purposes. The main idea behind these techniques is to interpret the program by simulating its execution with symbolic values rather than executing it on real hardware. Although, many symbolic execution techniques assume that the hardware does not experience errors during the execution of the program, the work [3] describes a framework which introduces a formal model to represent programs expressed in a generic assembly language with the ability to bring about faults that could potentially cause program failures.

Unlike symbolic methods, experimental measurement is an attractive option for evaluating an existing system or prototype closely in the field, because it allows the real execution of the system to be observed to obtain measurements

(hopefully highly accurate) in its working environment. In this regard, fault injection is an attractive option in verifying the fault tolerance requirements present in critical systems. Using SystemC Transaction Level Models (TLM) it is possible to model mixed hardware/software models in order to simulate the system in the presence of faults. For example, works [6, 7] use this methodology for the design and testing of fault tolerant systems implemented in an FPGA platform with different types of diagnostic techniques. The experimental results show the fault coverage and how Single Event Upset (SEU) occurrences cause faulty behaviours in the implemented systems. [8, 9, 10] use the same approach to verify the software of networked embedded systems long before the final hardware is available.

The work [11] describes a previous virtual platform from the Space Research Group of the University of Alcalá. The framework integrates a SPARC Instruction Set Simulator (ISS) together with other platform components by means of TLM 2.0 interfaces. It enables early software development and verification of platforms based on LEON3, a 32bit SPARC CPU-based system used by the European Space Agency. SimSoC [12] is a similar environment for ARM processors.

The framework presented in this work is a specific LEON2 virtual platform with fault injection capabilities for the Instrument Control Unit (ICU) of the Energetic Particle Detector (EPD) on board Solar Orbiter, along with the first test results of the boot software. As far as we know, the platform provides fault injection capabilities that had never been provided so far by any other LEON2 based development platform.

2. Solar Orbiter mission

Solar Orbiter [13] is a planned Sun-observation satellite, under development by the European Space Agency (ESA), which is scheduled to be launched in January 2017 as a baseline. At its closest point, around 0.3 Astronomical Units (AU) (see Figure 1), the spacecraft will be closer to the Sun than any previous

spacecraft. At about one-third of Earth's distance from the Sun, Solar Orbiter will be exposed to sunlight 13 times more intense than that on the Earth. The spacecraft must also withstand powerful bursts of atomic particles coming from the solar atmosphere. Solar Orbiter is intended to take detailed measurements of the inner heliosphere and nascent solar wind, and to perform close observations of the polar regions of the Sun, which is difficult to do from the Earth. The most accurate solar wind data obtained so far are those provided by the Solar and Heliospheric Observatory (SOHO) [14], a previous Sun-observation mission. SOHO was launched on December 2, 1995 and the SRG group was involved in the development of the Common Data Processing Unit (CDPU) for the COSTEP-ERNE Particle Analyzer Collaboration (CEPAC) instrument [15].

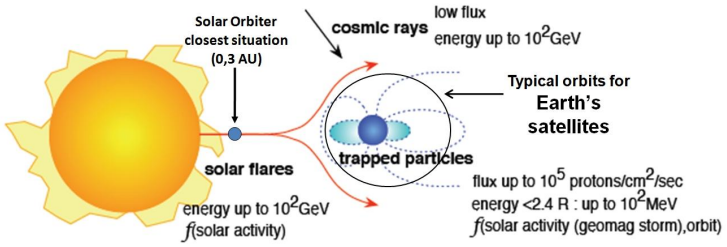


Figure 1: Solar Orbiter's relative situation to the Sun

Operating electronic devices in space leads to a high probability of random spurious defects caused by various radiation sources. Radiation-induced failures can be caused by Single Event Effects (SEEs), which are brought about by single protons or heavy ions hitting the electronic devices, or by the accumulated absorption of radiation, also known as a Total Ionizing Dose (TID), which leads to degradation over time. In the context of this paper we focus on the most likely case of SEEs causing probabilistic memory errors, namely transient Single Event Upsets (SEUs), which give rise to *flipped bits*, and potentially permanent failures like Single Event Latch-ups (SELs), which cause *stuck bits*, so they get a fixed logical value regardless of the attempts to change them [16].

2.1. Energetic Particle Detector

The Energetic Particle Detector [17] experiment will measure the composition, timing, and distribution functions of suprathermal and energetic particles. As shown in Figure 2, EPD consists of five separate sensors sharing a Common Data Processing Unit (CDPU) and a Low-Voltage Power Supply (LVPS) unit. Each sensor has a specific measurement to cover the required range of particles and energies. The complete set of sensors is as follows:

- The Electron Proton Telescope (EPT).
- The SupraThermal Electrons, Ions and Neutrals Telescope (STEIN).
- The Suprathermal Ion Spectrograph (SIS).
- The Low Energy Telescope (LET).
- The High Energy Telescope (HET).

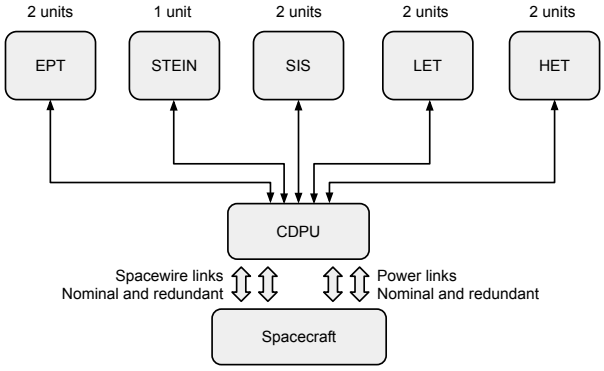


Figure 2: Solar Orbiter's EPD block diagram

The Space Research Group of the University of Alcalá is in charge of the development of the EPD's CDPU along with the corresponding boot and application software.

2.2. Space weather and effects on space software

Radiation-induced SEEs are a serious problem for spacecraft flight software, potentially leading to the complete loss of a mission. In order to analyse radiation effects from a software perspective, it is necessary to categorize the types of potential failures. SEEs can be classified as transient or permanent, correctable or uncorrectable, and affecting memory chips or processors. The SEUs of memory chip cells represent the most likely scenario, resulting in randomly *flipped bits* that can be corrected by rewriting the affected cells.

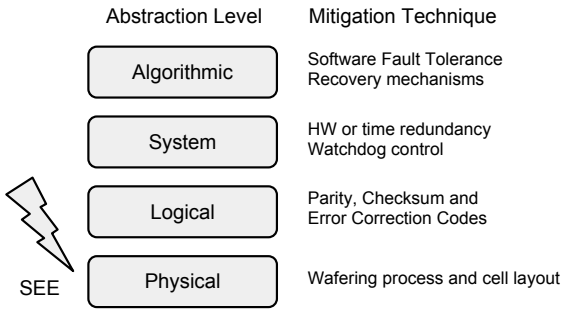


Figure 3: SEE mitigation techniques

As is highlighted in Figure 3, the first line in fault mitigation techniques relies on the chip manufacturing process and the hardware logic. In the work presented in this paper, the approach is to look at SEEs from a software perspective, and to design flight software explicitly so that it can detect and solve the majority of SELs. This type of *radiation-tolerant* flight software will significantly reduce the residual risks for critical missions. Because of the proximity of the Sun, from an on-board software designer perspective, it is mandatory to look out for permanent soft errors resulting from latch-up failures.

3. Instrument Control Unit software

The ICU software (ICUSW) is responsible for the EPD's command and data handling. It manages the system's start-up, the Telemetry and Telecommand (TM/TC) interfaces with the spacecraft, the interface with the sensors, the error handling and the data processing. Specifically, the boot software is in charge of the system's start-up stage. It manages the ICU configuration and carries out the EPD's overall status checking. The boot software of the ICU is always executed after a reset or a system power-on. The last stage of the boot software deploys and passes the control to the EPD's application software in accordance with the EPD's target mode.

The memory map is organized in three banks (see Figure 4), PROM, EEPROM and SDRAM. The PROM stores the boot code that, as well as the aforementioned functionality, it is in charge of carrying out sanity checks on the EEPROM and SDRAM. If an error is detected in the EEPROM, the CDPU has to be able to patch this memory from the boot code. This process can be achieved thanks to the ability of the boot code to support basic commanding with the spacecraft through the SpaceWire link. The EEPROM contains both the application software and the sensor's calibration tables. This memory area is implemented as two independent banks and stores two versions of the application software, namely the baseline and the updatable. The former is stored in a closed EEPROM bank, while the latter is stored in an opened and writable bank, allowing software updates during the mission. Finally, the SDRAM is used to store scientific and housekeeping data, tasks' stacks and heap. The SDRAM is protected against SEUs thanks to the use of an Error Detection and Correction (EDAC) mechanism, combined with memory scrubbing techniques.

One of the strictest requirements of the flight software is that the application binaries should be updatable in flight. This characteristic allows the system to recover itself from application bugs discovered after launch. In accordance with the ICUSW robustness requirements [18], the boot software takes control of the ICU automatically after a reset or a power-on and it is in charge of

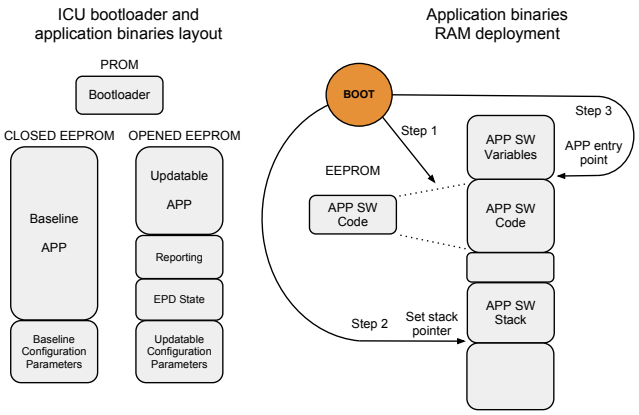


Figure 4: ICU boot and application software layout

deploying the main application software. The boot software, during the EPD's status checking stage, verifies the integrity of the two versions of the application software, as well as the integrity of the memory on which they are deployed. Depending on the result of this checking, it will deploy one of the two versions of the application software. If both application software images are damaged, the boot software will wait and remain active until the images have been repaired by means of service telecommands that allow memory patch, dump, and check operations. Each application software image is organized in segments whose integrity is checked by a Cyclic Redundancy Check (CRC) code, so that any possible damage in the EEPROM will only require the affected segments to be repaired.

The SDRAM will be tested in order to find possible failures resulting from `stuck-at` faults that may condition the system's stack allocation and application binary deployment. Again, when neither of the software application versions can be deployed, the ICU boot software must wait for spacecraft telecommands that will load a new version of the application software that avoids the damaged SDRAM areas. This behaviour specified in the ICUSW requirements

is summarized in Figure 5.

	Expected behaviour			
SDRAM baseline and updatable deployment addresses stuck fault	No APP boot, wait for spacecraft patch commands. Telemetry 53	No APP boot, wait for spacecraft patch commands. Telemetry 53	No APP boot, wait for spacecraft patch commands. Telemetry 53	No APP boot, wait for spacecraft patch commands. Telemetry 53, 54
SDRAM updatable deployment addresses stuck fault	Baseline version boot PC: 0x42000000. Telemetry: 51	No APP boot, wait for spacecraft patch commands. Telemetry 53	Baseline version boot PC: 0x42000000. Telemetry: 53, 51	No APP boot, wait for spacecraft patch commands. Telemetry 53, 54
SDRAM baseline deployment addresses stuck fault	Updatable version boot PC: 0x43000000. Telemetry: 51	Updatable version boot PC: 0x43000000. Telemetry: 51	No APP boot, wait for spacecraft patch commands. Telemetry 53	No APP boot, wait for spacecraft patch commands. Telemetry 53, 54
No corruption	Updatable version boot PC: 0x43000000. Telemetry: 51	Updatable version boot PC: 0x43000000. Telemetry: 51	Baseline version boot PC: 0x42000000. Telemetry: 53, 51	No APP boot, wait for spacecraft patch commands. Telemetry 53, 54
SDRAM / **EEPROM**	**No corruption**	**Baseline EEPROM binary corruption**	**Updatable EEPROM binary corruption**	**Baseline and updatable EEPROM binary corruption**

Figure 5: ICU boot faulty scenarios

3.1. ICU boot software development & verification challenges

The verification of the ICU boot software should be addressed at an early development stage because it affects the quality assurance of the overall on-board software. Moreover, the robustness requirements call for an exhaustive testing of the boot process and possible corruption of application binaries stored in the EEPROM or stuck-at faults in the SDRAM application deployment areas. Bearing this in mind, from a hardware dependant software point of view, such as the boot software, the major problem of carrying out early development and testing activities is the absence of a hardware platform on which to run it.

Other points to keep in mind are the effort and risks associated with bringing up hardware dependant software such as boot loaders. The classic approach for developing this kind of software has been to design the hardware, make a physical prototype, write the code, and then integrate the hardware and software. This methodology is nowadays too slow and calls for an alternative to the traditional *software-after-hardware* design flow in order to get started with software development and testing before the hardware is ready. To shortcut these issues the SRG has developed Leon2ViP, a LEON2 virtual platform with

fault-injection capabilities. Figure 6 shows both the timeline design flow with a classic approach and the flow using a virtual platform. For the ICU boot software development, the use of Leon2ViP has meant a significant development improvement and an eventually shorter system integration when the real hardware becomes available.

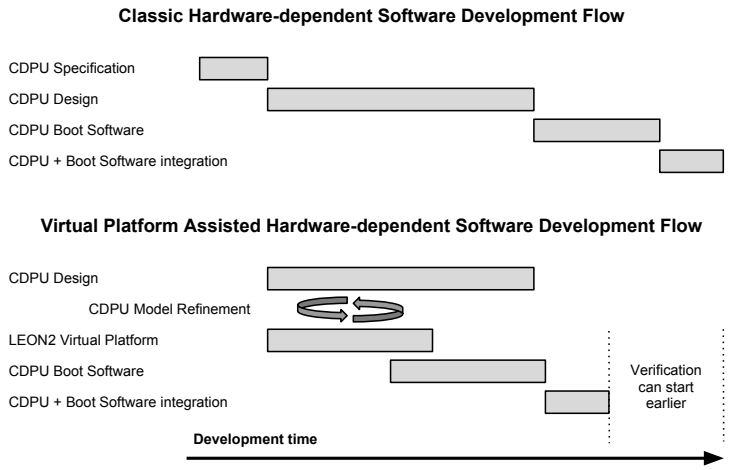


Figure 6: ICU software design flow

Another important point is the testing of recovery mechanisms. As is pointed out in [19], rarely executed code, like exception handling, is a significant factor in determining product quality: "The size of the [rarely used] code was 20% less than the [frequently used code], but it contributed 2.5 times more to the post-release failures that brought the system down."

As can be seen in Figure 7, most of the effort is devoted to functional requirements. This figure is an adaptation of failure distribution models taken from [20]. It plots the failure intensity due to undiscovered bugs versus the verification time progress. It also shows that high coverage figures does not always mean a better fault tolerance capabilities verification. Usually, robustness or fault tolerance requirements fall outside the classic testing procedures, making

fault injection capabilities necessary. This leads to the paradox that the procedures that have to handle exceptional situations are never tested. Again, a way to overcome this situation is through the use of virtual platforms.

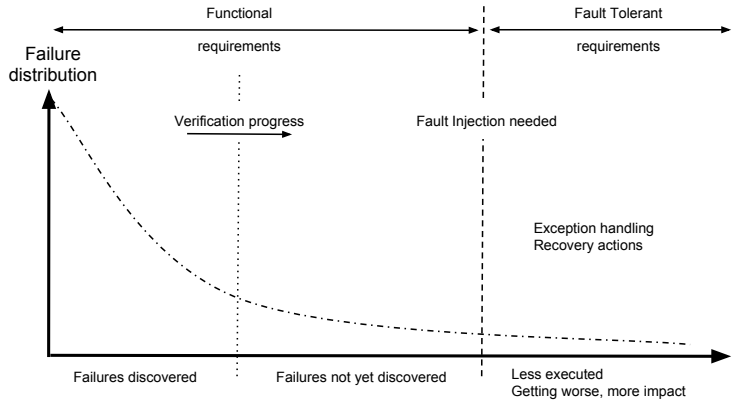


Figure 7: Fault tolerance requirements verification needs

With all these needs, the major challenges in ICU boot software development and testing are to:

- start the boot software development process when no ICU hardware was available;
- review the boot sequence as specified in the requirements, especially for the exception handling mechanisms when either the binary or deployment areas are corrupted;
- obtain a user friendly code coverage report of the boot software from the execution traces of all failure execution scenarios.

4. Leon2ViP Framework

For the development of the boot software of the ICU, it is advisable to have a framework able to run the same binary code as it will run on real hardware

and emulate the SDRAM and EEPROM permanent errors which is difficult, if not impossible, in real hardware. The solution comes from the use of a system-level modelling language to build a virtual platform. The Transaction Level Modeling (OSCI TLM), layered on top of the SystemC class library [21, 22] raises the abstraction level description of a system, focusing on the exchange of data between components through communication channels or sockets. In fact, the primary goal of TLM2 is to allow early software development and to join the hardware and software design flows together.

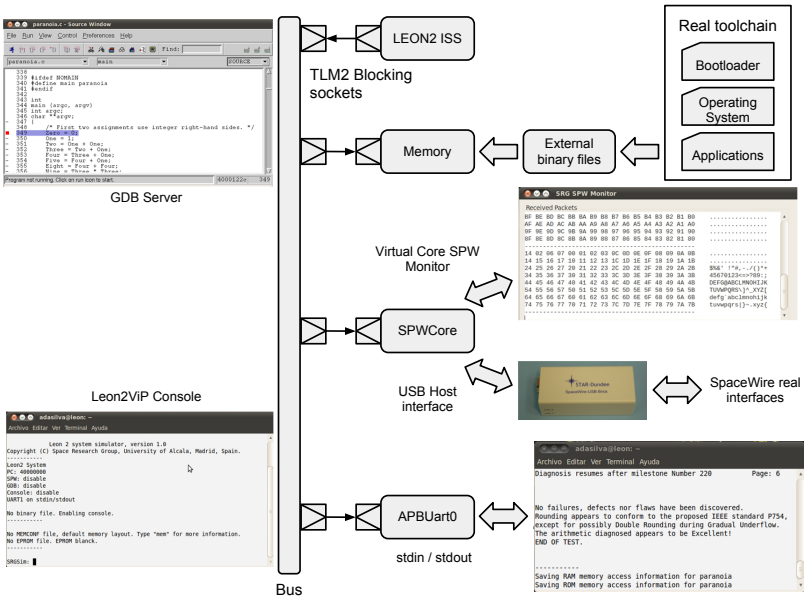


Figure 8: Leon2ViP Framework components

Leon2ViP is a transaction level model of a complete LEON2 system that is built around SystemC/TLM2 interfaces as is described in Figure 8. The use of SystemC provides an industry-standard mechanism to model and verify systems using standard C++ tools. Because of their advantages, TLM models have been traditionally used to design space exploration, early architectural performance

estimations and to allow an earlier start to software development, by joining the hardware and software design flows together. They also allow the emulation of permanent stuck-at faults in memory, which are difficult to emulate on real hardware. The main TLM2.0 components of Leon2ViP are:

- LEON2 ISS: SPARC V8 untimed Instruction Set Simulator with blocking TLM2 transaction interfaces. The code of this module has been rewritten taking in account the expertise of the SRG groups developing LEON3 systems.
- Memory: PROM, EEPROM and SDRAM blocks as were described in Figure 4. The memory layout is highly configurable through an external file and the current contents can be read from an external ordinary binary file generated by the compiler toolchain. This module has been completely written from the scratch.
- SPWCore: SpaceWire [23] interface for spacecraft on-board communications. This interface can be mapped onto a SpaceWire monitor or an external SpaceWire hardware based on a Star-Dundee USB SpaceWire brick [24]. This module has been completely written from the scratch.
- APBUart0: This interface is provided for serial communications. It is used to perform standard input/output and it can also be mapped onto a real host serial port. This module is practically the same as in LEON3 systems.

The contents of the memory layout description file for the CDPU system are as follows:

```
# Leon2ViP - Memory layout definition file
#
# Each line define a memory block, expected fields are:
# NAME: Memory Block Name, used by ‘‘mem’’ command
# START ADDRESS: Memory Block start address
# SIZE: Memory Block size in bytes
```

```
# ATTRIBUTES:
#     R (readable), W (writable), RW (both)
#     S (static) = contents are saved/loaded from a file named ‘‘name.bin’’

RAM    0x40000000 0x10000000 RW
EPROM0 0x20000000 0x00080000 RWS
EPROM1 0x30000000 0x00080000 RWS
ROM    0x00000000 0x00008000 R
```

RAM and ROM areas are usually filled with external program binary files using a load command. The memory blocks labelled with the static (S) attribute have a special behaviour. Although the regular load command also works with these memory areas, they are filled at the system start-up with the contents of a file which has the same name as the memory block. The file should have the same size as the memory block and must contain its binary image. No processing on the data stored in the files is carried out. When the execution ends, the contents of the memory blocks are written back to the files. This emulates the behaviour of an EEPROM when data updates are carried out on it.

For controlling the software execution, Leon2ViP offers a command-line interface with which the user can issue commands for loading memory binaries, for inspecting the internal state of the processor and the memory blocks and for defining breakpoints/watchpoints, among many others. A debugging server (GDB) is also included through a TCP/IP network connection.

Figure 9 shows two typical working sessions in which the capabilities of the virtual platform can be seen. The first one, starting from the left, shows the execution of an RTEMS application running three tasks. The second one shows the execution of an eCos operating system application. Both examples reflect the virtual platform capability to execute multitasking real time applications on-top of embedded operating systems.

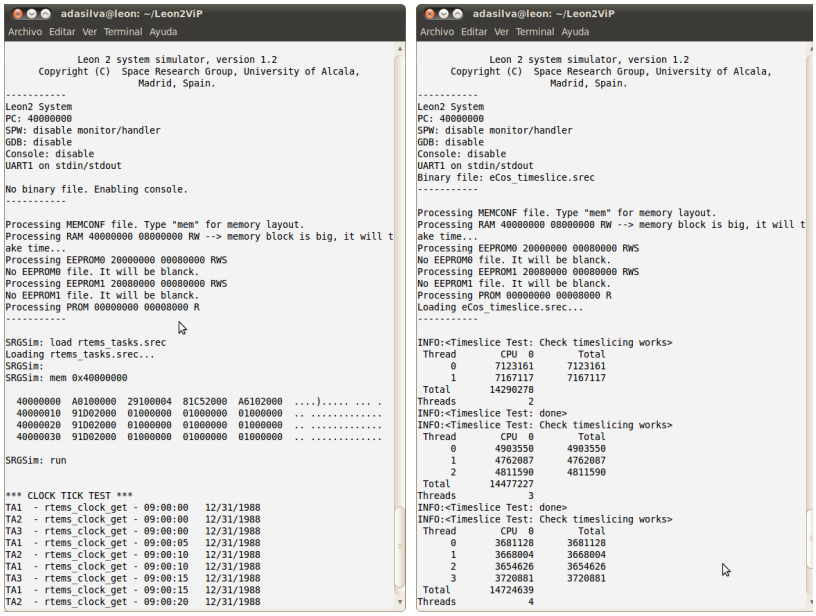


Figure 9: Leon2ViP Execution demos examples

4.1. Batch mode and fault injection capabilities

The commands mentioned in the previous section can be saved in a common plain ASCII file in order to be executed by the virtual platform without having to type them in every time. Additionally, Leon2ViP incorporates several memory corruption related commands. For this work the most significant is `stuckat0`. This command applies a fault mask to the content of a memory position using an AND operation. Unlike the `bitflip` command, the fault remains active until it is explicitly disabled by means of a `stuckend` command. This way, it is possible to simulate a temporary or permanent fault resulting from a burnout. Following is an example of a batch file. It loads a PROM binary, sets the program counter (PC) and the next program counter (NPC) to their initial values, sets breakpoints to SDRAM areas where the application binaries stored in EEPROM are going to be deployed, sets some stuck-at-zero faults at the

beginning of the updatable deployment SDRAM areas and runs the program.

```
load CDPU_Prom.srec
reg pc 0
reg npc 4
break 0x43000000  # breakpoint at updatable SDRAM deployment area
break 0x42000000  # breakpoint at baseline SDRAM deployment area

stuckat0 0x43000000 0xFFFF00FF    #stuck at zero fault at 0x43000002
#stuckat0 0x42000000 0x00000000    #stuck at zero fault at 0x42000000
#stuckat0 0x20000000 0x00000000    #stuck at zero fault at 0x20000000
#stuckat0 0x20080000 0x00000000    #stuck at zero fault at 0x20080000
continue
quit
```

The lines beginning with "#" are treated as comments. Depending on which fault is injected, the boot software code being tested must behave as described in Figure 5.

5. ICU boot software start-up tests and results

Figure 10 shows the methodology used to verify the fulfilment of the robustness requirements of the ICU boot software start-up.

Starting from the specification of the requirements, software engineers write boot software code to meet the requirements. From the requirements document, fault injection parameters are extracted, such as stuck-at places in the memory map and the test oracles. Test oracles are the conditions that should be met to decide whether an execution is successful or not. Both EEPROMs contain a basic application, binary-linked in different deployment areas. Initially, about 7 tests were performed manually, namely:

- No corruption at boot time.
- Baseline EEPROM corruption by means of a stuck-at fault at address 0x20000000. SDRAM is OK.

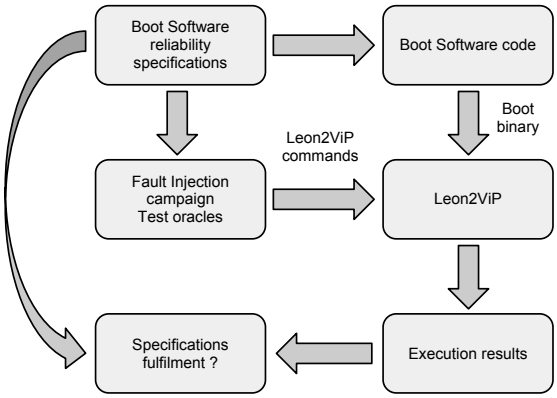


Figure 10: Leon2ViP fault injection procedure

- Updatable EEPROM corruption by means of a stuck-at fault at address 0x20080000. SDRAM is OK.
- Both Baseline and Updatable EEPROM corruption by means of stuck-at faults at addresses 0x20000000 and 0x20080000. SDRAM is OK.
- SDRAM Baseline deployment area corruption by means of a stuck-at fault at address 0x42000000. EEPROMs are OK.
- SDRAM Updatable deployment area corruption by means of a stuck-at fault at address 0x43000000. EEPROMs are OK.
- Both SDRAM Baseline and Updatable deployment areas corruption by means of stuck-at faults at addresses 0x42000000 and 0x43000000. EEPROMs are OK.

This test set covers all the possible corruption configurations for the starting address of each memory area. The tests were carried out successfully and the boot software behaved as expected. As an example, Figure 11 shows the execution results for the situation when both SDRAM deployment areas have a stuck-at-zero fault at the beginning of their corresponding memory areas. It

can be seen that Leon2ViP stops the execution at the address of the routine in which the boot software waits for spacecraft telecommands. Previously, it has sent two telemetry packets (of services 53 and 54) through the SpaceWire link. Service 5 of the ECSS Packet Utilisation Standard is used to report information of operational significance to the user. Subservices 3 and 4 are used to report anomalies or errors of medium and high severity respectively [25].

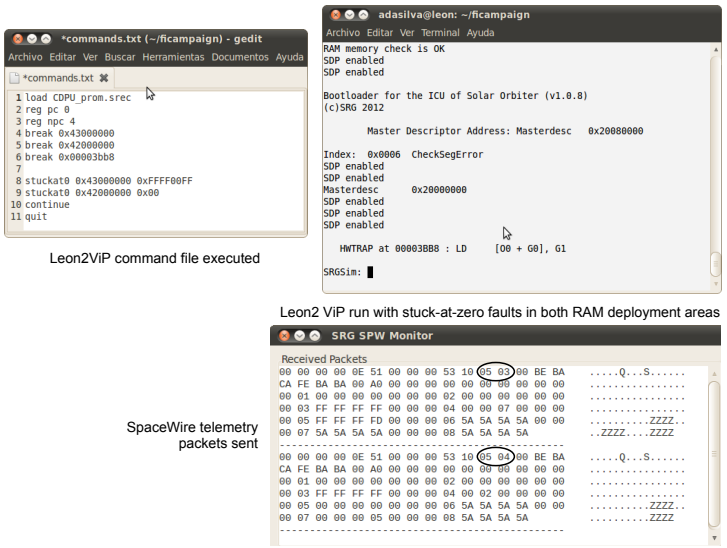


Figure 11: Leon2ViP Fault Injection Test

In order to perform a complete test, these corruption configurations must be applied to all significant memory locations. Below are the sizes of the memory sections of the updatable version of the application software. The same applies for the baseline version but starting at address 0x42000000.

```
Sections:
Idx  Name           Size      VMA       LMA       File off  Algn
0 .text         000350a0  43000000  43000000  00000060  2**3
                CONTENTS, ALLOC, LOAD, CODE
```

```
1 .data         00000f20  430350a0  430350a0  00035100  2**3
                CONTENTS, ALLOC, LOAD, DATA
2 .jcr          00000004  43035fc0  43035fc0  00036020  2**2
                CONTENTS, ALLOC, LOAD, DATA
3 .bss          00004a78  43035fd0  43035fd0  00036024  2**4
```

The sum of the sizes of all memory sections is 0x3AA3C bytes. So, one test run is necessary, inserting one stuck-at fault, at each and every memory location to carry out an exhaustive test of the SDRAM updatable deployment area. This should be repeated for each corruption previously described configuration. This leads to a total amount of 1,441,128 runs. Each boot takes around 3 seconds to complete so the entire test would take about 50 days running in a single machine to perform an exhaustive testing of the SDRAM deployment areas. The use of virtual platforms can significantly reduce the time spent since several instances of the Leon2ViP can be run in parallel on different real machines, thus shortening the overall testing time.

5.1. ICU boot software exhaustive fault campaign

A complete fault injection campaign has been carried out by sweeping corruption memory addresses from the beginning to the end of every section in both EEPROM application binaries and their corresponding SDRAM deployment areas. This is done by means of a simple shell script that generates a specific command file for each run. It launches Leon2ViP and redirects the standard output to a file and finally analyses the execution results. The following is a schematic description of the script for testing the updatable deployment area.

```
Updatable RAM Campaign

1  ini_address   0x43000000    # start updatable deployment area
2  end_address   0x4303AA3C    # end updatable binary
3
4  for address = ini_address; address <= end_address; address++
```

```
5
6       generate batch command file with breakpoints and  stuck-at faults
7       launch leon2_vip in batch mode with stdout redirected to file
8       analyze leon2_vip execution and write results to file
9
10 end for
```

For each execution, the behaviour of the boot software being tested is checked. There are three possibilities for each fault injected, as detailed in Figure 5:

- Updatable version launch.
- Baseline version launch.
- No application launch, wait for spacecraft commands.

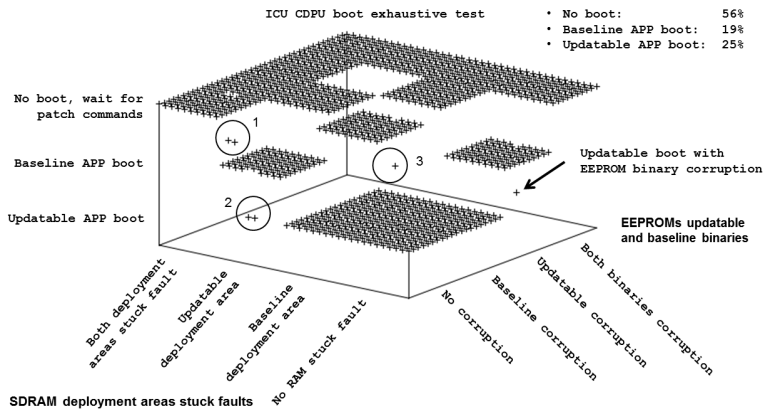


Figure 12: ICU boot software test executions

All boot software test executions and their corresponding application launches are shown in Figure 12. Although most of the runs behaved as expected, several special situations arose, which are highlighted by the numbers 1, 2, 3 in the figure. Another highlighted error corresponds to an updatable application launch

despite the binary corruption of its EEPROM image. Where do these abnormal executions come from and what are their causes? The fault model applied in this test is a simple stuck-at-zero on the content of every memory location. But what happens when the original value saved in memory *was already* zero? There are two possible situations:

- The value stored at the EEPROM position is zero. EEPROM content sanity check is based on a CRC checking schema. From the point of view of the CRC verification, there is no way of knowing whether the zero value is an original one or another brought about by a stuck-at fault: all it knows is that the CRC matches. In this case, no faulty behaviour is expected.
- The value deployed in the SDRAM is zero. In this case, an EEPROM zero value is going to be deployed in a SDRAM position with a stuck-at-zero fault. Since the basic SDRAM memory test done is a read-after-write verification, no memory fault is detected. From this, two complete different situations can be derived:
  - The SDRAM memory location is read-only data or a program operation code. It is assumed that the value is correct and it will never change. No side effects are expected.
  - The SDRAM memory location belongs to an ordinary variable with an initial value of zero. This is a dangerous situation because the program cannot update the variable value and therefore, when the variable is read back, a wrong value (zero) is obtained. From this point, the application behaviour is unpredictable and depends on variable functionality. This is an unacceptable behaviour and the solution asks for a more exhaustive memory test at system startup in order to find faulty SDRAM locations.

5.2. ICU boot code coverage analysis

"Untested code is the dark matter of software" [26]. If a code is not executed then there are zero probabilities of exposing any bugs it may have. Otherwise,

just because a code statement has been executed, it doesn't mean that all potential errors have been exposed. In truth, the value of code coverage analysis is the ability to identify areas of code that have not been previously exercised, especially recovery and exception handling code, in order to propose new tests to evaluate these areas. At least, it improves code confidence and helps to reduce risks.

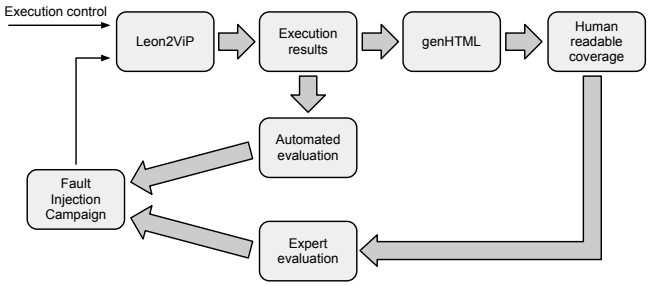


Figure 13: Leon2ViP code coverage

Figure 13 shows the Leon2ViP code coverage features. Each program execution can generate a file with all memory access information. From the code point of view, this information reflects which operation codes are executed and which are not. Given this information, a specific fault injection can be configured to exercise unexecuted code. In order to perform an expert evaluation, it is necessary to provide human-readable coverage information. An auxiliary application takes the execution traces from Leon2ViP and generates an HTML coverage report as shown in Figure 14.

6. Conclusions

The capability of injecting faults is essential for the verification of fault tolerance mechanisms that are envisaged in the construction of space systems, as well as predicting the consequences of bad system behaviour resulting from errors not detected in functional tests. The analysis of the experimental results

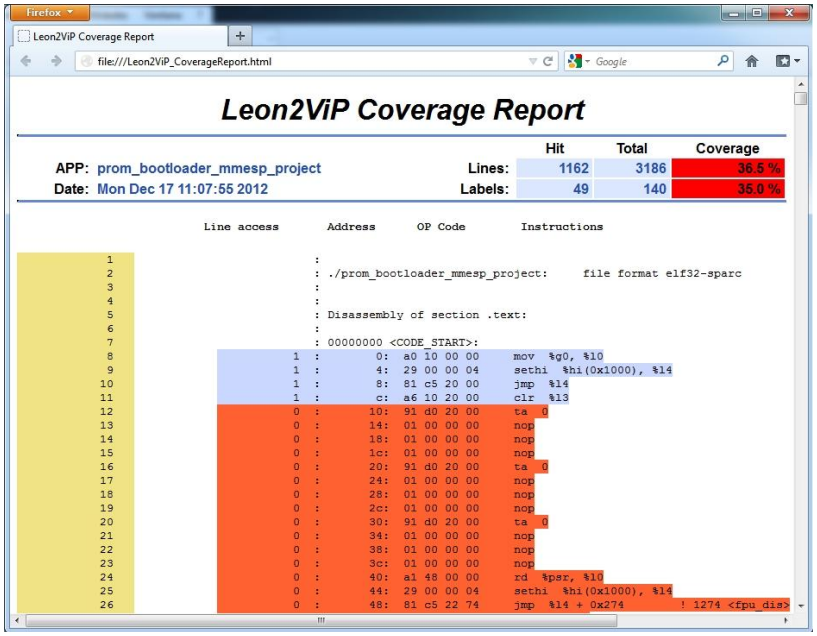


Figure 14: Leon2ViP Coverage report

indicates that it is feasible to identify strategic memory locations where faults have more catastrophic consequences and hence to improve the software fault tolerance.

For the development of the EPD's ICU embedded software, Leon2ViP provides fast editing, compile, debug cycles through more controllability, observability and determinism in the experiments carried out. It reduces the dependencies of software and system tasks on hardware availability, allowing an earlier boot and system software development and verification. All the faulty scenarios described in the requirements, even those involving permanent memory errors can be simulated. Even with real hardware available, Leon2ViP enables unmanned and tightly focused fault injection campaigns, not possible otherwise, in order to expose and diagnose flaws in the software implementation early. Furthermore,

the use of a virtual hardware-in-the-loop approach makes it possible to carry out preliminary integration tests with the spacecraft emulator or the sensors.

References

[1] Robillard M. P. and Murphy G. C. Static Analysis to Support the Evolution of Exception Structure in Object-oriented Systems. ACM Transactions on Software Engineering Methodology, 12(2):191-221, Apr. 2003.

[2] Voas J. and Charron F. and McGraw G. Predicting How Badly 'Good' Software Can Behave, IEEE Software, 14(4):73-83.

[3] Bailan, O., Rossi, U., Wantens, A., Daveau, JM., Nappi, S. and Roche, P. Verification of soft error detection mechanism through fault injection on hardware emulation platform, Dependable Systems and Networks Workshops (DSN2010), 2010

[4] Randimbivololona, F., Brahmiy, A. and Le Meurz, P. Airborne Software Tests on a Fully Virtual Platform, Ninth European Dependable Computing Conference - EDCC, 2012.

[5] Wang Y., Wang, L. and Zheng, Z. Application of Virtual Prototype Technology to Simulation Test for Airborne Software System Advances in Electronic Engineering, Communication and Management Vol.2 Lecture Notes in Electrical Engineering Volume 140, 2012, pp 653-658.

[6] Straka, M., Kastil, J., Kotasek, Z. and Miculka, L. Fault tolerant system design and SEU injection based testing. Microprocessors and Microsystems, Available online 8 September 2012, ISSN 0141-9331.

[7] Cuenca-Asensi, S., Martínez-Álvarez, A., Restrepo-Calle, F., Palomo, F. R., Guzmán-Miranda, H. and Aguirre, M. A. Soft core based embedded systems in critical aerospace applications Journal of Systems Architecture, Volume 57, Issue 10, November 2011, Pages 886-895, ISSN 1383-7621.

[8] Ziemke, C., Kuwahara, T. and Kossev, I. An integrated development framework for rapid development of platform-independent and reusable satellite on-board software, Acta Astronautica, Volume 69, Issues 7-8, September-October 2011, Pages 583-594, ISSN 0094-5765.

[9] Ginsberg, D., Mignogna, A., Carloni, M., Menichelli, F., Ferrari, A., Nguyen, D. and Scholte, E. SystemC based Simulation for Virtual Prototyping of Large Scale Distributed Embedded Control Systems 3nd International Workshop on Analysis Tools and Methodologies for Embedded and Realtime Systems (WATERS 2012), Pisa (Italy), July 2012.

[10] Weiyun L. and Radetzki, M. Concurrent and comparative fault simulation in SystemC and its application in robustness evaluation. Microprocessors and Microsystems, Available online 10 September 2012, ISSN 0141-9331.

[11] Da Silva, A., & Sánchez, S., LEON3 ViP: A Virtual Platform with Fault Injecion Capabilities, 13th Euromicro Conf. on Digital Systems Design: Architectures, Methods and Tools (DSD), 2010.

[12] Helmester, C. & Joloboff, V. SimSoC: A SystemC TLM integrated ISS for full system simulation, Proceedings of International Asia Pacific Conference on Computer Architecture and Systems, 2008.

[13] European Space Agency Solar Orbiter Exploring the Sun-heliosphere Definition study report, July 2011.

[14] Harboe-Sorensen, R., Daly, E., Teston F., Schweitzer, H., Nartallo, R., Perol, P., Vandenbussche, F., Dzitko, H., & Cretolle, J., Observation and Analysis of Single Event Effects On-Board the SOHO Satellite, IEEE Transactions on Nuclear Science, Vol. 49, No. 3, June 2002.

[15] Müller-Mellin, R. et. al. COSTEP - Comprehensive Suprathermal And Energetic Particle Analyser Solar Physics 162: 483-504, 1995.

[16] Michael Nicolaidis Soft Errors in Modern Electronic Systems Springer, 2011 - ISBN: 978-2-4419-6992-7.

[17] Rodríguez, Ó & Fernández, J., (Space Research Group, U. Alcalá), Solar Orbiter Energetic Particle Detector, Architecture Design Document, 2012.

[18] Polo, Ó & Parra, P., (Space Research Group, U. Alcalá) Solar Orbiter Energetic Particle Detector, Flight Software Requirements Document, 2012.

[19] Hecht, H., and Wallace, D., Towards more effective testing for high assurance systems. Proc. High Assurance Systems Engineering Conf., Washington, DC, August 1997.

[20] Lyu, Michael R., Handbook of software reliability engineering, McGraw-Hill, Inc., isbn: 0-07-039400-8

[21] Open SystemC Initiative, SystemC http://www.systemc.org (2012).

[22] Open SystemC Initiative, TLM2.0 http://www.systemc.org/downloads/standards/tlm20/ (2012).

[23] European Space Agency http://spacewire.esa.int (2012).

[24] STAR Dundee http://www.star-dundee.com (2012).

[25] European cooperation for space standardization (ECSS) "Ground systems and operations Telemetry and telecommand packet utilization ECSS E- 70-41A , January 2003

[26] Cedrik's Blog http://beust.com/weblog/2006/08/03/untested-code-is-the-dark-matter-of-software/